

Basic tutorial

Adam Kiezun

Corporation and others 2000, 2005. This page is made available under license. For full details see the [LEGAL](#) in the documentation bundle.

Table of Contents

<u>Basic tutorial</u>	1
<u>Preparing Eclipse</u>	2
<u>Verifying JRE installation and classpath variables</u>	2
<u>Java projects</u>	5
<u>Java builder</u>	6
<u>Build classpath</u>	7
<u>Classpath variables</u>	8
<u>Java development tools (JDT)</u>	9
<u>Debugger</u>	10
<u>Breakpoints</u>	11
<u>Adding breakpoints</u>	12
<u>Java perspectives</u>	13
<u>Java</u>	13
<u>Java Browsing</u>	13
<u>Java Type Hierarchy</u>	13
<u>Debug</u>	13
<u>Java views</u>	15
<u>Package Explorer view</u>	15
<u>Hierarchy view</u>	15
<u>Projects view</u>	15
<u>Packages view</u>	15
<u>Types view</u>	15
<u>Members view</u>	15
<u>Changing the appearance of the console view</u>	17
<u>Console view</u>	18
<u>Stepping through the execution of a Java program</u>	19
<u>Step over</u>	19
<u>Step into</u>	19
<u>Step into Selection</u>	19
<u>Step with filters</u>	19
<u>Run to return</u>	19
<u>Run to line</u>	20

Table of Contents

<u>Launching a Java program</u>	21
<u>Java editor</u>	22
<u>Opening an editor for a selected element</u>	23
<u>Using the Java editor</u>	24
<u>Generating getters and setters</u>	25
<u>Creating a new class in an existing compilation unit</u>	26
<u>Creating a new Java class</u>	27
<u>Creating Java elements</u>	28
<u>Creating a new Java project</u>	29
<u>Creating a Java project as its own source container</u>	30
<u>Creating a Java project with source folders</u>	31
<u>Creating a new source folder</u>	33
<u>Java Build Path page</u>	34
<u>Source tab</u>	34
<u>Projects tab</u>	34
<u>Libraries tab</u>	35
<u>Order and Export tab</u>	36
<u>Default output folder</u>	36
<u>File actions</u>	37
<u>New Java Project Wizard</u>	39
<u>Project name page</u>	39
<u>Java settings page</u>	39
<u>Java Compiler</u>	41
<u>General</u>	41
<u>JDK Compliance</u>	41
<u>Classfile generation</u>	41
<u>Building</u>	42
<u>General</u>	42
<u>Build path problems</u>	42
<u>Output folder</u>	43
<u>Errors/Warnings</u>	43
<u>Code style</u>	43
<u>Potential programming problems</u>	44

Table of Contents

<u>Java Compiler</u>	
<u>Name shadowing and conflicts</u>	44
<u>Name shadowing and conflicts</u>	45
<u>Unnecessary code</u>	45
<u>J2SE 5.0 options</u>	46
<u>Building a Java program</u>	49
<u>Building automatically</u>	50
<u>Building manually</u>	51
<u>Incremental build</u>	51
<u>Incremental project build</u>	51
<u>Clean and rebuild from scratch (full build)</u>	51
<u>Clean and rebuild selected projects</u>	51
<u>Working with build paths</u>	53
<u>Viewing and editing a project's Java build path</u>	54
<u>Adding a JAR file to the build path</u>	55
<u>Adding a library folder to the build path</u>	56
<u>Creating a new JAR file</u>	57
<u>Attaching source to a JAR file</u>	58
<u>Attaching source to a class path variable</u>	59
<u>Adding a classpath variable to the build path</u>	60
<u>Defining a classpath variable</u>	62
<u>Deleting a classpath variable</u>	63
<u>Classpath variables</u>	64
<u>Configurable variables</u>	64
<u>Reserved class path variables</u>	64
<u>Working with JREs</u>	65
<u>Adding a new JRE definition</u>	66
<u>Assigning the default JRE for the workbench</u>	67

Table of Contents

<u>Choosing a JRE for a launch configuration.....</u>	68
<u>Running and debugging.....</u>	69
<u>Remote debugging.....</u>	70
<u>Using the remote Java application launch configuration.....</u>	71
<u>Disconnecting from a VM.....</u>	72
<u>Debug view.....</u>	73
<u>Local debugging.....</u>	75
<u>Resuming the execution of suspended threads.....</u>	76
<u>Evaluating expressions.....</u>	77
<u>Suspending threads.....</u>	78
<u>Catching Java exceptions.....</u>	79
<u>Removing breakpoints.....</u>	80
<u>Enabling and disabling breakpoints.....</u>	81
<u>Applying hit counts.....</u>	82
<u>Setting method breakpoints.....</u>	83
<u>Breakpoints view.....</u>	84
<u>Managing conditional breakpoints.....</u>	85
<u>Views and editors.....</u>	86
<u>Changing the appearance of the Hierarchy view.....</u>	87
<u>Using the Hierarchy view.....</u>	88
<u>Opening a type hierarchy on a Java element.....</u>	89
<u>Changing new type hierarchy defaults.....</u>	90
<u>Opening a type hierarchy on the current text selection.....</u>	91

Table of Contents

<u>Opening a type hierarchy in the workbench.....</u>	<u>92</u>
<u>Opening a type hierarchy in its own perspective.....</u>	<u>93</u>
<u>Type Hierarchy view.....</u>	<u>94</u>
<u>Type Hierarchy tree pane toolbar buttons.....</u>	<u>94</u>
<u>Member list pane toolbar buttons.....</u>	<u>94</u>
<u>Java.....</u>	<u>96</u>
<u>Navigate actions.....</u>	<u>98</u>
<u>Package Explorer view.....</u>	<u>100</u>
<u>Toolbar buttons.....</u>	<u>101</u>
<u>Java element filters dialog.....</u>	<u>102</u>
<u>Filtering elements.....</u>	<u>103</u>
<u>Using the Package Explorer view.....</u>	<u>104</u>
<u>Showing and hiding elements.....</u>	<u>105</u>
<u>Showing and hiding system files.....</u>	<u>106</u>
<u>Showing and hiding CLASS files generated for inner types.....</u>	<u>107</u>
<u>Showing and hiding libraries.....</u>	<u>108</u>
<u>Showing single element or whole Java file.....</u>	<u>109</u>
<u>Java editor.....</u>	<u>110</u>
<u>Toolbar actions.....</u>	<u>110</u>
<u>Key binding actions.....</u>	<u>110</u>
<u>Viewing documentation and information.....</u>	<u>112</u>
<u>Viewing Javadoc information.....</u>	<u>113</u>
<u>Using content/code assist.....</u>	<u>114</u>
<u>Scrapbook.....</u>	<u>115</u>
<u>Creating a Java scrapbook page.....</u>	<u>116</u>

Table of Contents

<u>Java scrapbook page</u>	117
<u>Displaying the result of evaluating an expression</u>	118
<u>Executing an expression</u>	119
<u>Inspecting the result of evaluating an expression</u>	120
<u>Viewing runtime exceptions</u>	121
<u>Expressions view</u>	122
<u>New Java Scrapbook Page Wizard</u>	123
<u>Viewing compilation errors and warnings</u>	124
<u>Setting execution arguments</u>	125
<u>Creating a Java application launch configuration</u>	126
<u>Changing the active perspective when launching</u>	128
<u>Debug preferences</u>	130
<u>Preparing to debug</u>	132
<u>Run and debug actions</u>	133
<u>Java search tab</u>	135
<u>Search string</u>	135
<u>Search For</u>	135
<u>Limit To</u>	136
<u>Scope</u>	136
<u>Java search</u>	137
<u>Searching Java code</u>	138
<u>Conducting a Java search using pop-up menus</u>	139
<u>Search actions</u>	140
<u>Conducting a Java search using the Search dialog</u>	142
<u>Formatting Java code</u>	143

Table of Contents

<u>Setting code formatting preferences</u>	144
<u>Formatting files or portions of code</u>	145
<u>Source actions</u>	146
<u>Code Formatter</u>	149
<u>Java editor</u>	150
<u>Appearance and Navigation</u>	150
<u>Code assist</u>	151
<u>Syntax Coloring</u>	152
<u>List of Quick Assists</u>	154
<u>Quick Fix</u>	158
<u>JDT actions</u>	161
<u>Frequently asked questions on JDT</u>	162
<u>Can I use a Java compiler other than the built-in one (javac for example) with the workbench?</u>	162
<u>Where do Java packages come from?</u>	162
<u>When do I use an internal vs. an external JAR library file?</u>	162
<u>When should I use source folders within a Java project?</u>	162
<u>What are source attachments. How do I define one?</u>	162
<u>Why are all my resources duplicated in the output folder (bin, for example)?</u>	162
<u>How do I prevent having my documentation files from being copied to the project's output folder?</u>	163
<u>How do I create a default package?</u>	163
<u>What is refactoring?</u>	163
<u>When do I use code select/code resolve (F3)?</u>	163
<u>Is the Java program information (type hierarchy, declarations, references, for example) produced by the Java builder? Is it still updated when auto-build is off?</u>	163
<u>After reopening a workbench, the first build that happens after editing a Java source file seems to take a long time. Why is that?</u>	163
<u>I can't see a type hierarchy for my class. What can I do?</u>	163
<u>How do I turn off "auto compile" and do it manually when I want?</u>	164
<u>When I select a method or a field in the Outline view, only the source for that element is shown in the editor. What do I do to see the source of the whole file?</u>	164
<u>Can I nest source folders?</u>	164
<u>Can I have separate output folders for each source folder?</u>	164
<u>Can I have an output or source folder that is located outside of the workspace?</u>	164
<u>JDT glossary</u>	165
<u>Edit actions</u>	167

Table of Contents

<u>Using Quick Fix</u>	169
<u>Using Quick Assist</u>	170
<u>Quick fix</u>	171
<u>Java outline</u>	172
<u>Toolbar buttons</u>	172
<u>Restoring a deleted workbench element</u>	173
<u>Using the local history</u>	174
<u>Replacing a Java element with a local history edition</u>	175
<u>Comparing a Java element with a local history edition</u>	176
<u>Showing and hiding members</u>	177
<u>Appearance</u>	178
<u>Showing full or compressed package names</u>	179
<u>Showing and hiding override indicators</u>	180
<u>Showing and hiding method return types</u>	181
<u>Sorting elements in Java views</u>	182
<u>Java toolbar actions</u>	183
<u>New Java Package Wizard</u>	185
<u>Creating a new Java package</u>	186
<u>Moving folders, packages, and files</u>	187
<u>Refactoring support</u>	188
<u>Refactoring</u>	189
<u>Refactoring without preview</u>	191
<u>Refactoring with preview</u>	192
<u>Previewing refactoring changes</u>	193

Table of Contents

<u>Undoing a refactoring operation</u>	194
<u>Redoing a refactoring operation</u>	195
<u>Refactor actions</u>	196
<u>Using Structured Selection</u>	199
<u>Using Surround with Try/Catch</u>	200
<u>Extracting a method</u>	201
<u>Renaming a method</u>	202
<u>Renaming method parameters</u>	203
<u>Changing method signature</u>	204
<u>Refactoring Dialog</u>	205
<u>Wizard based refactoring user interface</u>	206
<u>Parameter pages</u>	206
<u>Preview page</u>	206
<u>Problem page</u>	206
<u>.JDT icons</u>	208
<u>Objects</u>	208
<u>Object adornments</u>	209
<u>Build path</u>	210
<u>Code assist</u>	211
<u>Compare</u>	211
<u>Debugger</u>	211
<u>Editor</u>	213
<u>JUnit</u>	213
<u>NLS tools</u>	214
<u>Quick fix</u>	214
<u>Refactoring</u>	215
<u>Search</u>	215
<u>Search – Occurrences in File</u>	215
<u>Type hierarchy view</u>	215
<u>Dialog based refactoring user interface</u>	217
<u>Input dialog</u>	217
<u>Preview dialog</u>	217
<u>Problem dialog</u>	217

Table of Contents

<u>Override methods</u>	219
<u>Extract method errors</u>	220
<u>Extracting a local variable</u>	222
<u>Inlining a local variable</u>	223
<u>Replacing a local variable with a query</u>	224
<u>Copying and moving Java elements</u>	225
<u>Extracting a constant</u>	227
<u>Renaming a package</u>	228
<u>Opening a package</u>	229
<u>Showing an element in the Package Explorer view</u>	230
<u>Renaming a compilation unit</u>	231
<u>Creating a new interface in an existing compilation unit</u>	232
<u>Creating a new Java interface</u>	233
<u>Creating a top-level interface</u>	234
<u>Creating a nested interface</u>	235
<u>Renaming a type</u>	236
<u>Creating a new Java enum</u>	237
<u>New Java Enum Wizard</u>	238
<u>Creating a new Java annotation</u>	239
<u>New Java Annotation Wizard</u>	240
<u>Creating a top-level class</u>	241
<u>Creating a nested class</u>	243
<u>New Java Class Wizard</u>	244

Table of Contents

<u>New Source Folder Wizard.....</u>	246
<u>New Java Interface Wizard.....</u>	247
<u>Opening a type in the Package Explorer view.....</u>	248
<u>Organizing existing import statements.....</u>	249
<u>Adding required import statements.....</u>	250
<u>Managing import statements.....</u>	251
<u>Setting the order of import statements.....</u>	252
<u>Organize Imports.....</u>	253
<u>Renaming a field.....</u>	254
<u>Renaming a local variable.....</u>	255
<u>Parameters page.....</u>	256
<u>Inlining a method.....</u>	257
<u>Inlining a constant.....</u>	258
<u>Self encapsulating a field.....</u>	259
<u>Pulling members up to superclass.....</u>	260
<u>Pushing members down to subclasses.....</u>	261
<u>Moving static members between types.....</u>	262
<u>Moving an instance method to a component.....</u>	263
<u>Converting a local variable to a field.....</u>	264
<u>Converting an anonymous inner class to a nested class.....</u>	265
<u>Converting a nested type to a top level type.....</u>	266
<u>Extracting an interface from a type.....</u>	267
<u>Replacing references to a type with references to one of its supertypes.....</u>	268

Table of Contents

<u>Replacing a single reference to a type with a reference to one of its supertypes.....</u>	<u>269</u>
<u>Replacing an expression with a method parameter.....</u>	<u>270</u>
<u>Replacing constructor calls with factory method invocations.....</u>	<u>271</u>
<u>Inferring type parameters for generic type references.....</u>	<u>272</u>
<u>Opening an editor on a type.....</u>	<u>273</u>
<u>Open Type.....</u>	<u>274</u>
<u>Project actions.....</u>	<u>275</u>
<u>Run menu.....</u>	<u>276</u>
<u>Content/Code Assist.....</u>	<u>277</u>
<u>Templates.....</u>	<u>278</u>
<u>Template dialog.....</u>	<u>278</u>
<u>Template variables.....</u>	<u>279</u>
<u>Templates.....</u>	<u>281</u>
<u>Using templates.....</u>	<u>282</u>
<u>Writing your own templates.....</u>	<u>284</u>
<u>Task Tags.....</u>	<u>285</u>
<u>Code templates.....</u>	<u>286</u>
<u>Code and Comments.....</u>	<u>286</u>
<u>Comment templates.....</u>	<u>286</u>
<u>New Java files template.....</u>	<u>286</u>
<u>Catch block body template.....</u>	<u>287</u>
<u>Method body template.....</u>	<u>287</u>
<u>Constructor body template.....</u>	<u>287</u>
<u>Getter body template.....</u>	<u>287</u>
<u>Setter body template.....</u>	<u>287</u>
<u>Code Template dialog.....</u>	<u>287</u>
<u>Code style.....</u>	<u>289</u>
<u>Naming Conventions.....</u>	<u>289</u>
<u>Code Conventions.....</u>	<u>289</u>
<u>Create Getters and Setters.....</u>	<u>291</u>

Table of Contents

<u>String externalization</u>	292
<u>Finding strings to externalize</u>	293
<u>Externalizing Strings</u>	294
<u>Finding unused and incorrectly used keys in property files</u>	295
<u>Using the Externalize Strings Wizard</u>	296
<u>Key/value page</u>	297
.....	298
<u>Property File page</u>	300
<u>Externalize Strings Wizard</u>	302
<u>String selection page</u>	302
<u>Translation settings page</u>	302
<u>Error page</u>	303
<u>Preview page</u>	303
<u>Viewing marker help</u>	304
<u>Javadoc location page</u>	305
<u>Javadoc generation</u>	306
<u>First page</u>	306
<u>Standard doclet arguments</u>	306
<u>General arguments</u>	307
<u>Creating Javadoc documentation</u>	309
<u>Selecting types for Javadoc generation</u>	310
<u>Configuring Javadoc arguments for standard doclet</u>	312
<u>Configuring Javadoc arguments</u>	314
<u>Showing and hiding empty packages</u>	316
<u>Showing and hiding empty parent packages</u>	317
<u>Showing and hiding Java files</u>	318
<u>Showing and hiding non-Java elements</u>	319

Table of Contents

<u>Showing and hiding non-Java projects</u>	320
<u>Showing and hiding import declarations</u>	321
<u>Showing and hiding package declarations</u>	322
<u>Finding overridden methods</u>	323
<u>Display view</u>	325
<u>Variables view</u>	326
<u>Show detail pane</u>	327
<u>Show detail pane</u>	328
<u>Re-launching a program</u>	329
<u>Console preferences</u>	330
<u>JRE installations</u>	331
<u>Source attachments</u>	332
<u>JAR</u>	332
<u>Variable</u>	332
<u>Editing a JRE definition</u>	334
<u>Deleting a JRE definition</u>	335
<u>Overriding the default system libraries for a JRE definition</u>	336
<u>Installed JREs</u>	337
<u>Defining the JAR file's manifest</u>	338
<u>Creating a new manifest</u>	338
<u>Using an existing manifest</u>	338
<u>Setting advanced options</u>	340
<u>JAR file exporter</u>	341
<u>JAR package specification</u>	341
<u>JAR packaging options</u>	341
<u>JAR manifest specification</u>	342
<u>Creating JAR files</u>	343

Table of Contents

<u>Regenerating a JAR file.....</u>	344
<u>Adding source code as individual files.....</u>	345
<u>From a ZIP or JAR file.....</u>	345
<u>From a directory.....</u>	345
<u>Adding a JAR file as a library.....</u>	347
<u>Java Compiler page.....</u>	348
<u>Converting line delimiters.....</u>	349
<u>Finding and replacing.....</u>	350
<u>Using the Find/Replace dialog.....</u>	351
<u>Using Incremental Find.....</u>	352
<u>Finding next or previous match.....</u>	353
<u>Changing the encoding used to show the source.....</u>	354
<u>Commenting and uncommenting lines of code.....</u>	355
<u>Shifting lines of code left and right.....</u>	356
<u>Exclusion and inclusion filters.....</u>	357
<u>Access rules.....</u>	358
<u>Creating a new source folder with exclusion filter.....</u>	359
<u>Starting from scratch.....</u>	359
<u>From an existing Java Project.....</u>	359
<u>Creating a new source folder with specific output folder.....</u>	361
<u>Creating your first Java project.....</u>	362
<u>Getting the Sample Code (JUnit).....</u>	362
<u>Creating the project.....</u>	362
<u>Browsing Java elements using the package explorer.....</u>	366
<u>Opening a Java editor.....</u>	368
<u>Adding new methods.....</u>	371

Table of Contents

<u>Using content assist</u>	374
<u>Identifying problems in your code</u>	376
<u>Using code templates</u>	379
<u>Organizing import statements</u>	382
<u>Using the local history</u>	384
<u>Extracting a new method</u>	386
<u>Creating a Java class</u>	390
<u>Renaming Java elements</u>	398
<u>Moving and copying Java elements</u>	401
<u>Navigate to a Java element's declaration</u>	403
<u>Viewing the type hierarchy</u>	406
<u>Searching the workbench</u>	412
<u>Performing a Java search from the workbench</u>	412
<u>Searching from a Java view</u>	414
<u>Searching from an editor</u>	414
<u>Continuing a search from the search view</u>	415
<u>Performing a file search</u>	416
<u>Viewing previous search results</u>	417
<u>Running your programs</u>	419
<u>Debugging your programs</u>	424
<u>Evaluating expressions</u>	429
<u>Evaluating snippets</u>	431
<u>Notices</u>	433
<u>About This Content</u>	433
<u>License</u>	434
<u>Using the Java browsing perspective</u>	435
<u>Writing and running JUnit tests</u>	437
<u>Writing Tests</u>	437
<u>Running Tests</u>	438
<u>Customizing a Test Configuration</u>	439
<u>Debugging a Test Failure</u>	440

Table of Contents

<u>Writing and running JUnit tests</u>	
<u>Creating a Test Suite</u>	440
<u>Project configuration tutorial</u>	442
<u>Detecting existing layout</u>	443
<u>Layout on file system</u>	443
<u>Steps for defining a corresponding project</u>	443
<u>Sibling products in a common source tree</u>	447
<u>Layout on file system</u>	447
<u>Steps for defining corresponding projects</u>	447
<u>Organizing sources</u>	452
<u>Layout on file system</u>	452
<u>Steps for defining a corresponding project</u>	452
<u>Overlapping products in a common source tree</u>	459
<u>Layout on file system</u>	459
<u>Steps for defining corresponding "Product1" and "Product2" projects</u>	459
<u>Product with nested tests</u>	467
<u>Layout on file system</u>	467
<u>Steps for defining a corresponding project</u>	467
<u>Products sharing a common source framework</u>	475
<u>Layout on file system</u>	475
<u>Steps for defining corresponding projects</u>	475
<u>Nesting resources in output directory</u>	483
<u>Layout on file system</u>	483
<u>Steps for defining a corresponding project</u>	483
<u>Project using a source framework with restricted access</u>	493
<u>Layout on file system</u>	493
<u>Steps for defining corresponding projects</u>	493
<u>Getting Started with Eclipse 3.1 and J2SE 5.0</u>	504
<u>Prerequisites</u>	504
<u>Compiler Compliance Level</u>	504
<u>Generic Types</u>	506
<u>Annotations</u>	508
<u>Enumerations</u>	509
<u>Autoboxing</u>	509
<u>Enhanced for loop</u>	510
<u>Other</u>	510

Table of Contents

<u>Creating a new Java Scrapbook Page</u>	512
<u>Parameters page</u>	513
<u>Problems page</u>	514
<u>Parameters page</u>	515
<u>Parameters page</u>	516
<u>Parameters page</u>	517
<u>Parameters page</u>	518
<u>Parameters page</u>	519
<u>Parameters page</u>	520
<u>Parameters page</u>	521
.....	522
<u>Parameters page</u>	523
.....	524
<u>Parameters page</u>	525
.....	526
<u>Parameters page</u>	527
.....	528
.....	529
<u>Parameters page</u>	531
.....	532
<u>Building circular projects</u>	533
<u>Building without cleaning the output location</u>	534
<u>Attaching source to a library folder</u>	535

Table of Contents

<u>Launching a Java applet</u>	536
<u>Launching a Java program in debug mode</u>	537
<u>Inspecting values</u>	538
<u>Using code assist</u>	539
<u>Scrapbook error reporting</u>	540
<u>Viewing compilation errors</u>	541
<u>Go to file for breakpoint</u>	542
<u>Add Java exception breakpoint</u>	543
<u>Suspend policy</u>	544
<u>Hit count</u>	545
<u>Uncaught</u>	546
<u>Caught</u>	547
<u>Modification</u>	548
<u>Access</u>	549
<u>Exit</u>	550
<u>Entry</u>	551
<u>Select all</u>	552
<u>Enable</u>	553
<u>Disable</u>	554
<u>Remove selected breakpoint</u>	555
<u>Remove all breakpoints</u>	556
<u>Show qualified names</u>	557
<u>Show supported breakpoints</u>	558

Table of Contents

<u>Properties</u>	559
<u>Copy</u>	560
<u>Select all</u>	561
<u>Find/Replace</u>	562
<u>Go to line</u>	563
<u>Clear</u>	564
<u>Terminate</u>	565
<u>Inspect</u>	566
<u>Display</u>	567
<u>Clear the display</u>	568
<u>Select all</u>	569
<u>Copy variables</u>	570
<u>Remove selected expressions</u>	571
<u>Remove all expressions</u>	572
<u>Change variable value</u>	573
<u>Show constants</u>	574
<u>Show static fields</u>	575
<u>Show qualified names</u>	576
<u>Show type names</u>	577
<u>Add/Remove watchpoint</u>	578
<u>Inspect</u>	579
<u>Open declared type</u>	580
<u>Show qualified names</u>	581

Table of Contents

<u>Show type names</u>	582
<u>Add/Remove watchpoint</u>	583
<u>Change variable value</u>	584
<u>Inspect</u>	585
<u>Step commands</u>	586
<u>JUnit</u>	587
<u>Java Task Tags page</u>	588
<u>Java Build Path page</u>	589
<u>Source tab</u>	589
<u>Projects tab</u>	590
<u>Libraries tab</u>	590
<u>Order and Export tab</u>	591
<u>Default output folder</u>	591
<u>Refactoring</u>	593
<u>Tips and Tricks</u>	593
<u>Editing source</u>	593
<u>Searching</u>	608
<u>Code navigation and reading</u>	610
<u>Java views</u>	616
<u>Miscellaneous</u>	621
<u>Debugging</u>	626
<u>What's New in 3.1</u>	637
<u>J2SE 5.0</u>	637
<u>Java Debugger</u>	652
<u>Java Compiler</u>	656
<u>Java Editor</u>	659
<u>General Java Tools</u>	665

Basic tutorial

This tutorial provides a step by step walk-through of the Java development tools.

Preparing Eclipse

In this section, you will verify that Eclipse is properly set up for Java development.

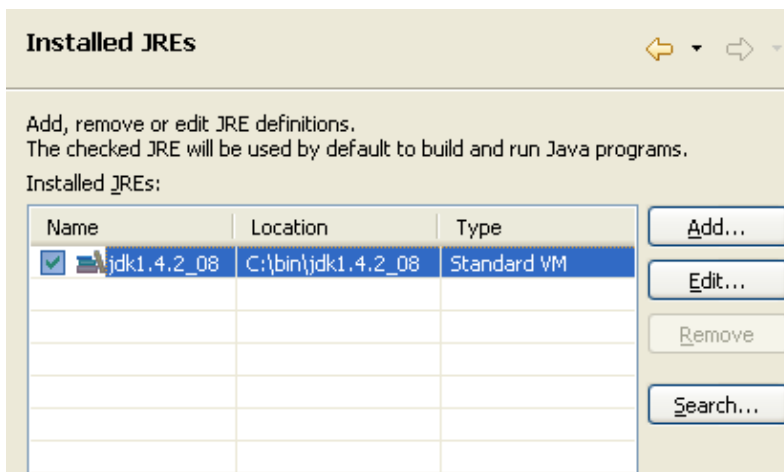
The following is assumed:

- You are starting with a new Eclipse installation with default settings.
- You are familiar with the basic Eclipse workbench mechanisms, such as views and perspectives.

If you're not familiar with the basic workbench mechanisms, please see the Getting Started chapter of the Workbench User Guide.

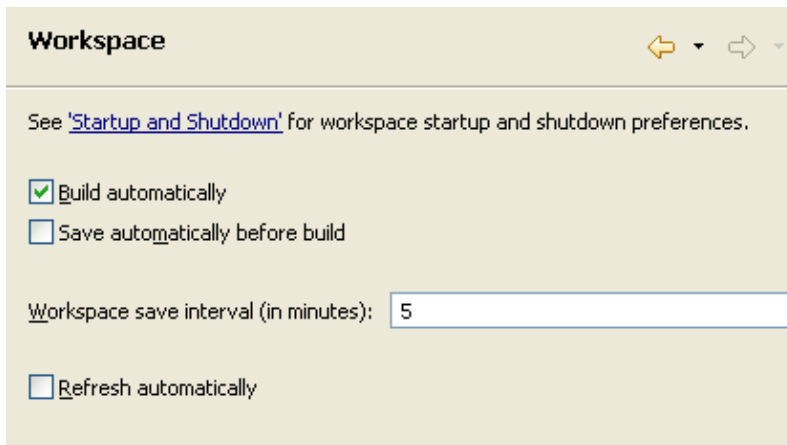
Verifying JRE installation and classpath variables

1. If you still see the Eclipse Welcome page, click the arrow icon to begin using Eclipse.
2. Select the menu item **Window > Preferences** to open the workbench preferences.
3. Select **Java > Installed JREs** in the tree pane on the left to display the *Installed Java Runtime Environments* preference page. Confirm that a JRE has been detected. By default, the JRE used to run the workbench will be used to build and run Java programs. It should appear with a checkmark in the list of installed JREs. We recommend that you use a Java SDK instead of a JRE. An SDK is designed for development and contains the source code for the Java library, easing debugging. Additional SDKs can be added by searching the hard drive for installed SDKs. To do so, simply click the *Search* button and specify a root folder for the search.



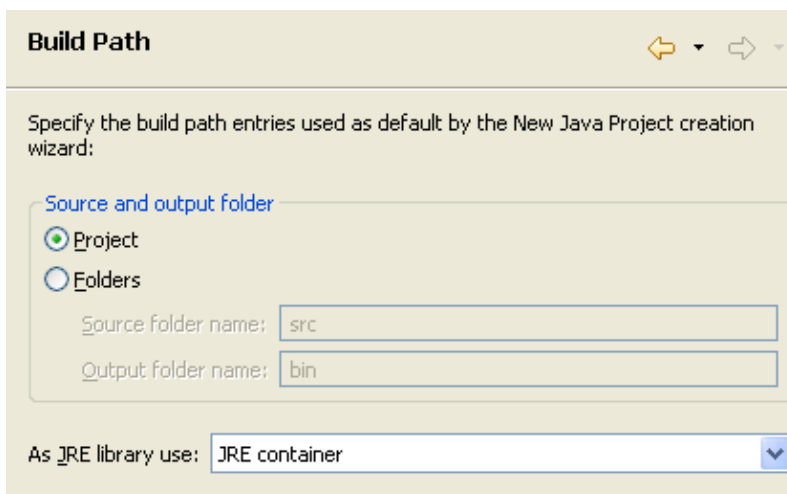
4. Select **General > Workspace** in the tree pane to display the *Workspace* preference page. Confirm that the *Build automatically* option is checked.

Basic tutorial



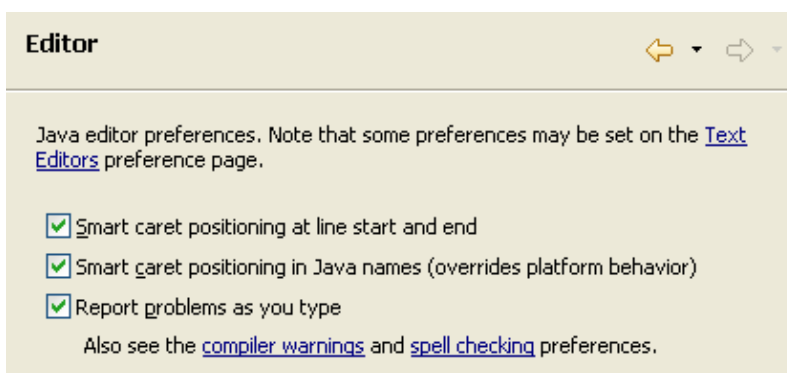
The screenshot shows the 'Workspace' preference page in Eclipse. At the top, there's a title bar with 'Workspace' and navigation arrows. Below the title bar, a message says 'See [\'Startup and Shutdown\'](#) for workspace startup and shutdown preferences.' There are three checkboxes: 'Build automatically' (checked), 'Save automatically before build' (unchecked), and 'Refresh automatically' (unchecked). A text field for 'Workspace save interval (in minutes):' contains the value '5'.

5. Select **Java > Build Path** in the tree pane to display the **Build Path** preference page. Confirm that **Source and output folder** is set to **Project**.



The screenshot shows the 'Build Path' preference page. The title bar has 'Build Path' and navigation arrows. The main text says 'Specify the build path entries used as default by the New Java Project creation wizard:'. There's a section titled 'Source and output folder' with two radio buttons: 'Project' (selected) and 'Folders'. Below these are two text fields: 'Source folder name:' with 'src' and 'Output folder name:' with 'bin'. At the bottom, there's a dropdown menu for 'As JRE library use:' set to 'JRE container'.

6. Select **Java > Editor** in the tree pane to display the **Java Editor** preference page. Confirm that option **Report problems as you type** is checked.



The screenshot shows the 'Editor' preference page. The title bar has 'Editor' and navigation arrows. The main text says 'Java editor preferences. Note that some preferences may be set on the [Text Editors](#) preference page.' There are three checkboxes: 'Smart caret positioning at line start and end' (checked), 'Smart caret positioning in Java names (overrides platform behavior)' (checked), and 'Report problems as you type' (checked). Below these, it says 'Also see the [compiler warnings](#) and [spell checking](#) preferences.'

7. Click on **OK** to save the preferences.

■ Related concepts

[Java projects](#)
[Classpath variables](#)
[Build classpath](#)

■ Related tasks

[Working with build paths](#)

[Working with JREs](#)

■ Related reference

[JRE Installations Preferences](#)

[Java Editor Preferences](#)

Java projects

A Java project contains source code and related files for building a Java program. It has an associated Java builder that can incrementally compile Java source files as they are changed.

A Java project also maintains a model of its contents. This model includes information about the type hierarchy, references and declarations of Java elements. This information is constantly updated as the user changes the Java source code. The updating of the internal Java project model is independent of the Java builder; in particular, when performing code modifications, if auto-build is turned off, the model will still reflect the present project contents.

You can organize Java projects in two different ways:

- Using the project as the source container. This is the recommended organization for simple projects.
- Using source folders inside the project as the source container. This is the recommended organization for more complex projects. It allows you to subdivide packages into groups.

■ Related concepts

[Java builder](#)

[Refactoring support](#)

■ Related tasks

[Creating a new Java project](#)

[Creating a Java project as its own source container](#)

[Creating a Java project with source folders](#)

[Creating a new source folder](#)

[Creating Java elements](#)

[Copying and moving Java elements](#)

[Generating getters and setters](#)

[Filtering elements](#)

■ Related reference

[New Java Project wizard](#)

Java builder

The Java builder builds Java programs using a compiler that implements the Java Language Specification. The Java builder can build programs incrementally as individual Java files are saved.

Problems detected by the compiler are classified as either warnings or errors. The existence of a warning does not affect the execution of the program; the code executes as if it were written correctly. Compile-time errors (as specified by the Java Language Specification) are always reported as errors by the Java compiler. For some other types of problems you can, however, specify if you want the Java compiler to report them as warnings, errors or to ignore them. To change the default settings, use the Window > Preferences > Java > Compiler > Errors/Warnings preference page.

The Java compiler can create CLASS files even in presence of compilation errors. However, in the case of serious errors (for example, references to inconsistent binaries, most likely related to an invalid build path), the Java builder does not produce any CLASS files.

■ Related concepts

[Build classpath](#)

[Java development tools \(JDT\)](#)

■ Related tasks

[Building a Java program](#)

[Building automatically](#)

[Building manually](#)

[Viewing compilation errors and warnings](#)

[Working with build paths](#) [Viewing and editing a project's build path](#)

[Adding a JAR file to the build path](#)

[Adding a library folder to the build path](#)

■ Related reference

[Java Build Path properties](#)

[Java Compiler preferences](#)

Build classpath

The build classpath is the path which is used to find classes that are referenced by your source code. During compilation, this path is used to search for classes outside of your project. The build classpath is specified for each project. In the project properties, it is referred to as the "Java Build Path."

■ Related concepts

[Java builder](#)

[Classpath variable](#)

[Exclusion and inclusion filters](#)

[Access rules](#)

■ Related tasks

[Adding a JAR file to the build path](#)

[Adding a library folder to the build path](#)

[Building a Java program](#)

[Building automatically](#)

[Building manually](#)

[Viewing and editing a project's build path](#)

[Working with build paths](#)

[Creating a new source folder with exclusion filter](#)

[Creating a new source folder with specific output folder](#)

■ Related reference

[Classpath Variables preferences](#)

[Java Build Path properties](#)

Classpath variables

The build path for a Java project can include source code files, other Java projects, and JAR files. JAR files can be specified using file system paths, or by using variables that refer to locations on the network.

Classpath variables allow you to avoid references to the location of a JAR file on your local file system. By using a classpath variable, you can specify a JAR file or library using only a variable name, such as JRE_LIB, rather than specifying the location of the JRE on your workstation. In this way, you can share build paths across teams and define the variables to refer to the correct location for your particular computer.

■ Related concepts

[Java development tools \(JDT\)](#)

[Build classpath](#)

■ Related tasks

[Adding a variable classpath entry](#)

[Attaching source to a classpath variable](#)

[Defining a classpath variable](#)

[Deleting a classpath variable](#)

■ Related reference

[Classpath Variables preferences](#)

[Java Build Path properties](#)

Java development tools (JDT)

The Java development tools (JDT) are a set of extensions to the workbench that allow you to edit, compile, and run Java programs.

■ Related concepts

[Build classpath](#)

[Classpath variables](#)

[Debugger](#)

[Java builder](#)

[Java editor](#)

[Java projects](#)

[Java perspectives](#)

[Java views](#)

[Java search](#)

[Refactoring support](#)

[Scrapbook](#)

■ Related tasks

[Adding source code as individual files](#)

[Creating Java elements](#)

[Formatting Java code](#)

[Restoring a deleted workbench element](#)

[Showing and hiding files](#)

[Working with JREs](#)

■ Related reference

[JDT actions](#)

[Frequently asked questions on JDT](#)

[JDT glossary](#)

Debugger

The JDT includes a debugger that enables you to detect and diagnose errors in your programs running either locally or remotely.

The debugger allows you to control the execution of your program by setting breakpoints, suspending launched programs, stepping through your code, and examining the contents of variables.

The debugger has a client/server design so you can debug programs running remotely on other systems in the network as well as programs running locally on your workstation. The debug client runs inside the workbench on your workstation. The debugger server runs on the same system as the program you want to debug. This could be a program launched on your workstation (local debugging) or a program started on a computer that is accessible through a network (remote debugging).

■ Related concepts

[Java development tools \(JDT\)](#)

[Breakpoints](#)

[Remote debugging](#)

[Local debugging](#)

■ Related tasks

[Adding breakpoints](#)

[Changing debugger launch options](#)

[Connecting to a remote VM with the Remote Java application launch configuration](#)

[Disconnecting from a VM](#)

[Evaluating expressions](#)

[Launching a Java program](#)

[Preparing to debug](#)

[Resuming the execution of suspended threads](#)

[Running and debugging](#)

[Suspending threads](#)

■ Related reference

[Debug preferences](#)

[Debug view](#)

[Run and debug actions](#)

Breakpoints

A breakpoint causes the execution of a program thread to suspend at the location where the breakpoint is set.

Breakpoints can be enabled and disabled via their context menus in the Breakpoints view.

- When a breakpoint is enabled, it will cause a thread to suspend whenever the breakpoint is reached. Enabled breakpoints are indicated with a blue circle. Enabled breakpoints are shown with a checkmark overlay after their class is loaded by the VM and the breakpoint is successfully installed.
- When a breakpoint is disabled, it will not cause threads to suspend. Disabled breakpoints are indicated with a white circle.

Breakpoints are displayed in the vertical editor ruler and in the Breakpoints view.

■ Related tasks

[Adding breakpoints](#)

[Resuming the execution of suspended threads](#)

[Running and debugging](#)

[Suspending threads](#)

■ Related reference

[Debug preferences](#)

[Debug view](#)

[Run menu](#)

[Run and debug actions](#)

[Breakpoints view](#)

Adding breakpoints

Line breakpoints are set on an executable line of a program.

1. In the editor area, open the file where you want to add the breakpoint.
2. Directly to the left of the line where you want to add the breakpoint, open the marker bar (vertical ruler) pop-up menu and select ***Toggle Breakpoint***. You can also double-click on the marker bar next to the source code line. A new breakpoint marker appears on the marker bar, directly to the left of the line where you added the breakpoint. Also, the new breakpoint appears in the Breakpoints view list.

While the breakpoint is enabled, thread execution suspends before that line of code is executed. The debugger selects the thread that has suspended and displays the stack frames on that thread's stack. The line where the breakpoint was set is highlighted in the editor in the Debug perspective.

■ Related concepts

[Debugger](#)

[Java perspectives](#)

[Java editor](#)

■ Related tasks

[Applying hit counts](#)

[Catching Java exceptions](#)

[Removing breakpoints](#)

[Enabling and disabling breakpoints](#)

[Managing conditional breakpoints](#)

[Setting method breakpoints](#)

[Stepping through the execution of a program](#)

■ Related reference

[Breakpoints view](#)

Java perspectives

The Java development tools contribute the following perspectives to the workbench:

Java

A perspective designed for working with Java projects. It consists of an editor area and the following views:

- Package Explorer
- Hierarchy
- Outline
- Search
- Console
- Tasks

Java Browsing

A perspective designed for browsing the structure of Java projects. It consists of an editor area and the following views:

- Projects
- Packages
- Types
- Members

Java Type Hierarchy

A perspective designed for exploring a type hierarchy. It can be opened on types, compilation units, packages, projects or source folders and consists of the Hierarchy view and an editor.

Debug

A perspective designed for debugging your Java program. It includes an editor area and the following views.

- Debug
- Breakpoints
- Expressions
- Variables
- Display
- Outline
- Console

Related concepts

[Java development tools \(JDT\)](#)

[Java views](#)

■ Related tasks

[Adding breakpoints](#)

[Opening a type hierarchy in its own perspective](#)

[Suspending threads](#)

■ Related reference

[Breakpoints view](#)

[Console view](#)

[Debug view](#)

[Display view](#)

[Expressions view](#)

[Java outline](#)

[Package Explorer view](#)

[Type Hierarchy view](#)

[Variables view](#)

Java views

The Java development tools contribute the following views to the workbench:

Package Explorer view

The Package Explorer view shows the Java element hierarchy of the Java projects in your workbench. It provides you with a Java–specific view of the resources shown in the Navigator. The element hierarchy is derived from the project's build class paths.

For each project, its source folders and referenced libraries are shown in the tree. You can open and browse the contents of both internal and external JAR files.

Hierarchy view

The Hierarchy view allows you to look at the complete hierarchy for a type, only its subtypes, or only its supertypes.

Projects view

The Projects view shows Java projects, source folders, external and internal libraries.

Note: source folders and libraries (both internal and external) presented in this view are not expandable. When they are selected, their contents are shown in the Packages view.

Packages view

The Packages view shows a list of Java packages from the currently selected Java projects, source folders or libraries. Typically, the Projects view is used to make this selection.

Types view

The Types view shows a list of Java types from the currently selected packages. Typically, the Packages view is used to make this selection.

Members view

The Members shows the content of a type, compilation unit or CLASS file. Typically, the Types view is used to make this selection.

Related concepts

[Java perspectives](#)

Related tasks

[Changing the appearance of the Console view](#)

[Changing the appearance of the Hierarchy view](#)

■ Related reference

[Breakpoints view](#)

[Console view](#)

[Debug view](#)

[Display view](#)

[Expressions view](#)

[Java outline](#)

[Package Explorer view](#)

[Type Hierarchy view](#)

[Variables view](#)

[Views and editors](#)

Changing the appearance of the console view

To set the types of output (and their colors) in the Console view:

1. From the menu bar, select **Window > Preferences > Debug > Console** to view the Console Preferences page.
2. Checking the **Show when program writes to standard out** checkbox will make the Console view visible each time new output is written to the console from the program's standard output stream. If there is no Console view in the current perspective, one will be created.
3. Checking the **Show when program writes to standard err** checkbox will make the Console view visible each time new output is written to the console from the program's standard error stream. If there is no Console view in the current perspective, one will be created.
4. Click any of the color buttons to change the color for the corresponding text stream.

To set the fonts used in the Console view:

1. From the menu bar, select **Window > Preferences > General > Appearance > Colors and Fonts** to view the Fonts Preferences page.
2. Select **Debug Console Text Font** from the list of fonts and use the **Change...** button to change the font. (The **Detail Pane Text Font** can be used to change the font of the debugger's detail pane).

■ Related concepts

[Debugger](#)
[Java views](#)

■ Related reference

[Console view](#)
[Views and editors](#)

Console view

This view shows the output of a process and allows you to provide keyboard input to a process. The console shows three different kinds of text, each in a different color.

- Standard output
- Standard error
- Standard input

You can choose the different colors for these kinds of text on the preferences pages (*Window > Preferences > Debug > Console*).

■ Related concepts

[Java views](#)

[Java perspectives](#)

■ Related tasks

[Changing the appearance of the console view](#)

[Stepping through the execution of a program](#)

[Run menu](#) [Breakpoints view](#) [Views and editors](#)

Stepping through the execution of a Java program

When a thread is suspended, the step controls can be used to step through the execution of the program line-by-line. If a breakpoint is encountered while performing a step operation, the execution will suspend at the breakpoint and the step operation is ended.

Step over

1. Select a stack frame in the Debug view. The current line of execution in that stack frame is highlighted in the editor in the Debug perspective.
2. Click the **Step Over** button in the Debug view toolbar, or press the **F6** key. The currently-selected line is executed and suspends on the next executable line.

Step into

1. Select a stack frame in the Debug view. The current line of execution in the selected frame is highlighted in the editor in the Debug perspective.
2. Click the **Step Into** button in the Debug view toolbar, or press the **F5** key. The next expression on the currently-selected line to be executed is invoked, and execution suspends at the next executable line in the method that is invoked.

Step into Selection

1. Select a stack frame in the Debug view. The current line of execution in the selected frame is highlighted in the editor in the Debug perspective.
2. In the Java editor, within the current line of execution, place the cursor on the name of a method that you would like to step into.
3. Click the **Step into Selection** action in the Run menu or Java editor context menu, or press the **Ctrl-F5** key. Execution resumes until the selected method is invoked.

Step with filters

1. Toggle the **Use Step Filters** button in the Debug view toolbar, or use **Shift+F5**. When the action is toggled on, each of the step actions (over, into, return) will apply the set of step filters which are defined in the user preferences (see **Window > Preferences > Java > Debug > Step Filtering**). When a step action is invoked, stepping will continue until an unfiltered location is reached or a breakpoint is encountered.

Run to return

1. Select a stack frame in the Debug view. The current line of execution in the selected frame is highlighted in the editor in the Debug perspective.
2. Click the **Run to Return** button in the Debug view toolbar or press the **F7** key. Execution resumes until the next return statement in the current method is executed, and execution suspends on the next executable line.

Run to line

When a thread is suspended, it is possible to resume execution until a specified line is executed. This is a convenient way to suspend execution at a line without setting a breakpoint.

1. Place your cursor on the line at which you want the program to run.
2. Select **Run to Line** from the pop-up menu or use **Ctrl+R**. Program execution is resumed and suspends just before the specified line is to be executed.

It is possible that the line will never be hit and that the program will not suspend.

Breakpoints and exceptions can cause the thread to suspend before reaching the specified line.

■ Related concepts

[Breakpoints](#)

[Java perspectives](#)

■ Related tasks

[Adding breakpoints](#)

[Launching a Java program](#)

[Resuming the execution of suspended threads](#)

[Running and debugging](#)

[Setting execution arguments](#)

[Suspending threads](#)

■ Related reference

[Debug view](#)

Launching a Java program

The simplest way to launch a Java program is to run it using a **Java Application** launch configuration. This launch configuration type uses information derived from the workbench preferences and your program's Java project to launch the program.

1. In the Package Explorer, select the Java compilation unit or class file you want to launch.
2. From the pop-up menu, select **Run > Java Application**. Alternatively, select **Run > Run As > Java Application** in the workbench menu bar, or Select **Run As > Java Application** in the drop-down menu on the **Run** tool bar button.
3. Your program is now launched, and text output is shown in the Console.

You can also launch a Java program by selecting a project instead of the compilation unit or class file. You will be prompted to select a class from those classes that define a *main* method. (If only one class with a main method is found in the project, that class is launched as if you selected it.)

■ Related concepts

[Java views](#)

[Java editor](#)

[Debugger](#)

■ Related tasks

[Connecting to a remote VM with the Java Remote Application launcher](#)

[Re-launching a program](#)

[Running and debugging](#)

[Setting execution arguments](#)

[Stepping through the execution of a program](#)

■ Related reference

[Debug view](#)

[Package Explorer](#)

Java editor

The Java editor provides specialized features for editing Java code.

Associated with the editor is a Java-specific Outline view, which shows the structure of the active Java compilation unit. It is updated as the user edits the compilation unit.

The Java editor can be opened on binary CLASS files. If a JAR file containing the CLASS files has a source attachment, then the editor shows the corresponding source.

The editor includes the following features:

- Syntax highlighting
- Content/code assist
- Code formatting
- Import assistance
- Quick fix
- Integrated debugging features

The Java editor can be configured to either show an entire compilation unit or a single Java element only. To change the setting, use the toolbar button Show Source of Selected Element Only.

The most common way to invoke the Java editor is to open a Java file from the Navigator or Package explorer using pop-up menus or by clicking the file (single or double-click depending on the user preferences). You can also open the editor by opening Java elements, such as types, methods, or fields, from other views.

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Opening an editor for a selected element](#)

[Using the Java editor](#)

[Using content/code assist](#)

[Formatting Java code](#)

[Adding required import statements](#)

[Generating getters and setters](#)

[Viewing compilation errors and warnings](#)

[Viewing runtime exceptions](#)

[Evaluating expressions](#)

■ Related reference

[Java editor actions](#)

[Java editor preferences](#)

[Java outline](#)

[Views and editors](#)

Opening an editor for a selected element

You can select the name of a type, method, or field in the Java source editor or in the scrapbook and open an editor on the definition of the element.

1. In the Java editor, select the name of a type, method, or field. You can also just click into the name once.
2. Do one of the following:
 - ◆ From the menu bar, select *Navigate > Open Declaration*
 - ◆ From the editor's pop-up menu, select *Open Declaration*
 - ◆ Press **F3**

or

1. Hold down the **Ctrl** key.
2. In the Java editor, move the mouse over the name of a type, method, or field until the name becomes underlined.
3. Click the hyperlink once.

If there are multiple definitions of the same name, a dialog is shown, and you can select one definition that you want to open. An editor opens containing the selected element.

■ Related concepts

Java editor

■ Related tasks

Using the Java editor

■ Related reference

Navigate menu

Views and editors

Using the Java editor

Note: Keyboard shortcuts used in the section (Java editor–related tasks) are the default key bindings. You can change between the *Standard* and the *Emacs* key binding sets by using the *Active configuration* combo box on *Window > Preferences > General > Keys*.

■ Related concepts

[Java editor](#)

■ Related tasks

[Generating getters and setters](#)

[Managing import statements](#)

[Using the local history](#)

[Formatting Java code](#)

[Viewing documentation and information](#)

[Using templates](#)

[Writing your own templates](#)

[Converting line delimiters](#)

[Finding and replacing](#)

[Changing the encoding used to show the source](#)

[Using quick fix](#)

[Using Structured Selection](#)

[Commenting and uncommenting lines of code](#)

[Shifting lines of code left and right](#)

[Using Surround with try/catch](#)

[Showing single elements or whole Java files](#)

[Opening an editor for a selected element](#)

[Using content/code assist](#)

■ Related reference

[Java Editor](#)

[Java outline](#)

[Java editor actions](#)

Generating getters and setters

The Java editor allows you to generate accessors ("getters and setters") for the fields of a type inside a compilation unit.

1. In the editor, select the field for which you want to generate accessors (or a type in which you want to create these methods).
2. Select ***Generate Getter and Setter*** from the *Source* pop-up menu.
3. A dialog will open to let you select which methods you want to create.
4. Select the methods and press OK.

■ Related concepts

[Java editor](#)

[Java projects](#)

■ Related tasks

[Using the Java editor](#)

[Creating a class in an existing compilation unit](#)

■ Related reference

[Generate Getter and Setter](#)

[Java outline](#)

Creating a new class in an existing compilation unit

An alternative way to create a new class is to add it to an existing compilation unit.

1. In the Package Explorer, double-click a compilation unit to open it in an editor.
2. Type the code for the class at the desired position in the compilation unit.

■ Related concepts

[Java projects](#)

■ Related tasks

[Creating a new Java class](#)

[Creating a top-level class](#)

[Creating a nested class](#)

[Renaming a class, field, or interface](#)

[Renaming a compilation unit](#)

[Setting execution arguments](#)

■ Related reference

[Package Explorer](#)

Creating a new Java class

Use the New Java Class wizard to create a new Java class. There are a number of ways to open this wizard:

1. Select the container where you want the new class to reside.
2. Click the **New Java Class** button in the workbench toolbar.

or

1. Select the container where you want the new class to reside.
2. From the container's pop-up menu, select **New > Class**.

or

1. Select the container where you want the new class to reside.
2. From the drop-down menu on the **New** button in the workbench toolbar, select **Class**.

or

1. Click the **New** button in the workbench toolbar to open the **New** wizard.
2. Select **Class** or **Java > Class** and click **Next**.

or

1. Select the container where you want the new class to reside.
2. Then, select from the menu bar **File > New > Class**.

■ Related concepts

[Java projects](#)

■ Related tasks

[Creating Java elements](#)

[Creating a new Java project](#)

[Creating a top-level class](#)

[Creating a nested class](#)

[Creating a class in an existing compilation unit](#)

[Setting execution arguments](#)

■ Related reference

[New Java Project wizard](#)

[New Source Folder wizard](#)

[New Java Package wizard](#)

[New Java Class wizard](#)

[Java Toolbar actions](#)

Creating Java elements

■ Related concepts

[Java projects](#)

[Java development tools \(JDT\)](#)

■ Related tasks

[Organizing Java projects](#)

[Creating a Java project as its own source container](#)

[Creating a Java project with source folders](#)

[Creating a new source folder](#)

[Creating a new Java package](#)

[Creating a new Java class](#)

[Creating a new Java enum](#)

[Creating a new Java interface](#)

[Creating a new Java annotation](#)

[Creating a Java scrapbook page](#)

[Adding a variable class path entry](#)

[Copying and moving Java elements](#)

[Defining a class path variable](#)

■ Related reference

[New Java Project wizard](#)

[New Source Folder wizard](#)

[New Java Package wizard](#)

[New Java Class wizard](#)

[New Java Enum wizard](#)

[New Java Interface wizard](#)

[New Java Annotation wizard](#)

[New Scrapbook Page wizard](#)

[Java Toolbar actions](#)

Creating a new Java project

You can organize Java projects in two different ways.

- Use the project as the container of packages. In this organization, all Java packages are created directly inside the project. This is the selected organization by default. The generated CLASS files are stored along with the JAVA source files.
- Use source folders as the container for packages. In this project organization, packages are not created directly inside the project but in source folders. You create source folders as children of the project and create your packages inside these source folders.

The default organization for new projects can be changed on the preference pages (**Window > Preferences > Java > Build Path**).

■ Related concepts

[Java projects](#)

■ Related tasks

[Creating a Java project as its own source container](#)

[Creating a Java project with source folders](#)

[Creating a new Java class](#)

[Creating a new Java interface](#)

[Creating a new source folder](#)

[Creating Java elements](#)

[Working with build paths](#)

■ Related reference

[New Java Project wizard](#)

Creating a Java project as its own source container

For simple projects, the project itself acts as the source container.

1. From the main workbench window, click **File > New > Project**. The New Project wizard opens.
2. Select **Java Project**, then click **Next**. The New Java Project wizard opens.
3. In the **Project name** field, type a name for your new Java project.
4. Select a location for the project and configure the JDK Compliance.
5. In the **Project layout** section, make sure **Use project folder as root for source and class files** is selected.
6. Click **Next**. The Java Settings Page opens.
7. On the **Source** tab, check that the project is the only source folder and the default output folder.
8. Optionally, on the **Projects** tab, select the required projects to be on the build path for this project. Use these options only if your project depends on other projects.
9. Optionally, on the **Libraries** tab, select JAR files and CLASS folders to add to the build path for this new project and attach source to the JAR files. Use these options only if your project requires additional libraries.
10. On the **Order and Export** tab, use the **Up** and **Down** buttons to move the selected JAR file or CLASS folders up or down in the build path order for this new project.
11. Click **Finish** when you are done.

■ Related concepts

[Java projects](#)

■ Related tasks

[Creating Java elements](#)

[Creating a new Java project](#)

[Creating a Java project with source folders](#)

[Creating a new Java package](#)

[Creating a new Java class](#)

[Creating a new interface in a compilation unit](#)

[Working with build paths](#)

■ Related reference

[New Java Project wizard](#)

Creating a Java project with source folders

For larger projects, create a project with source folders.

*Note: When using source folders, non-Java resources are copied to the output folder by the Java builder. If you have non-Java resources (documentation, for example) that should not be copied into the output folder, you can create an ordinary folder and store the resources there. You can also use the preferences page **Window > Preferences > Java > Compiler > Building** to specify a list of resources that will not be automatically copied to the output folder.*

1. From the main workbench window, click **File > New > Project**. The New Project wizard opens.
2. Select **Java Project**, then click **Next**. The New Java Project wizard opens.
3. In the **Project name** field, type a name for your new Java project.
4. Select a location for the project and configure the JDK Compliance.
5. In the **Project layout** section, make sure **Create separate source and output folders** is selected.
6. Click **Next**. The Java Settings Page opens.
7. On the **Source** tab, check that the project contains a source folder. You can create additional source folders at any time later.
8. Optionally, replace the default name in the **Default output folder** field to use a different name for the output folder.
9. On the **Projects** tab, select the required projects to be on the build path for this project.
10. On the **Libraries** tab, select JAR files and CLASS file containers to add to the build path for this new project and attach source to the JAR files.
11. On the **Order and Export** tab, use the **Up** and **Down** buttons to move the selected JAR file or CLASS file container up or down in the build path order for this new project.
12. Click **Finish** when you are done.

Note: When you are using CVS as your repository, it is recommended that you create a .cvsignore file inside the project. In this file, add a line with the name of the output folder ("bin" for example). Adding the output folder in the .cvsignore file ensures that the CVS versioning support ignores the output folder in versioning operations.

■ Related concepts

[Java projects](#)

[Java builder](#)

■ Related tasks

[Creating Java elements](#)

[Creating a new Java project](#)

[Creating a new source folder](#)

[Creating a Java project as its own source container](#)

[Creating a new Java package](#)

[Creating a new Java class](#)

[Creating a Java scrapbook page](#)

[Working with build paths](#)

■ Related reference

[New Java Project wizard](#)

Creating a new source folder

You can create a new folder to contain Java source code using the New Source Folder wizard.

1. In the Package Explorer, select the project where you want the new source folder to reside.
2. From the project's pop-up menu, select **New > Source Folder**. The New Source Folder wizard opens.
3. In the **Project Name** field, the name of the selected project appears. If you need to edit this field, you can either type a path or click **Browse** to choose a project that uses source folders.
4. In the **Folder Name** field, type a name for the new source folder.
5. If the new folder nests with an existing source folder you can check **Update exclusion filters in other source folders to solve nesting**. Otherwise you have to use the [Java Build Path](#) page (**Project > Properties > Java Build Path**) to fix the nesting conflict by removing other source folders.
6. Click **Finish** when you are done.

■ Related concepts

[Java projects](#)

■ Related tasks

[Creating Java elements](#)

[Creating a new Java project](#)

[Creating a new Java package](#)

[Creating a Java project as its own source container](#)

[Creating a Java project with source folders](#)

■ Related reference

[Java Build Path](#)

[New Source Folder wizard](#)

[Java Toolbar actions](#)

[Package Explorer](#)

Java Build Path page

The options in this page indicate the build path settings for a Java project. You can reach this page through the project properties (Project > Properties > Java Build Path) from the context menu on a created project or the [File menu](#) of the workbench.

The build class path is a list of paths visible to the compiler when building the project.

Source tab

Source folders are the root of packages containing .java files. The compiler will translate the contained files to .class files that will be written to the output folder. The output folder is defined per project except if a source folder specifies an own output folder. Each source folder can define an exclusion filter to specify which resources inside the folder should not be visible to the compiler.

Resources existing in source folders are also copied to the output folder unless the setting in the [Compiler preference page](#) (Window > Preferences > Java > Compiler > Building) specifies that the resource is filtered.

Source folder options

Option	Description
Add Folder	Creates a new folder to contain source
Edit	Allows to modify the currently selected source folder or source folder attribute.
Remove	Removes the selected folders from the class path. This does not delete the folders nor their contents.
Allow output folder per source folder	Shows/Hides the 'output folder' attribute of the source folders

Source folder attributes

Attribute	Description
Exclusion filter	Selects which resources are not visible to the compiler
Output folder	Only available when Allow output folder per source folder is checked. Defines a source folder specific output location. If not set the project's default output folder is used.

Projects tab

In the **Required projects on the build path** list, you can add project dependencies by selecting other workbench projects to add to the build path for this new project. The **Select All** and **Deselect All** buttons can be used to add or remove all other projects to or from the build path.

Adding a required project indirectly adds all its classpath entries marked as 'exported'. Setting a classpath entry as exported is done in the Order and Export tab.

The projects selected here are automatically added to the referenced projects list. The referenced project list is used to determine the build order. A project is always build after all its referenced projects are built.

Libraries tab

On this page, you can add libraries to the build path. You can add:

- Workbench–managed (internal) JAR files
- File system (external) JAR files
- Folders containing CLASS files
- Predefined libraries like the JRE System Library

JAR files can also be added indirectly as class path variables.

By default, the library list contains an entry representing the Java runtime library. This entry points to the JRE selected as the default JRE. The default JRE is configured in the [Installed JREs preferences page](#) (Window > Preferences > Java > Installed JREs)

Libraries tab options

Option	Description
Add JARs	Allows you to navigate the workbench hierarchy and select JAR files to add to the build path.
Add External JARs	Allows you to navigate the file system (outside the workbench) and select JAR files to add to the build path.
Add Variable	Allows you to add classpath variables to the build path. Classpath variables are an indirection to JARs with the benefit of avoiding local file system paths in a classpath. This is needed when projects are shared in a team. Variables can be created and edited in the Classpath Variable preference page (Window > Preferences > Java > Build Path > Classpath Variables)
Add Library	Allows to add a predefined libraries like the JRE System Library. Such libraries can stand for an arbitrary number of entries (visible as children node of the library node)
Add Class Folder	Allows to navigate the workbench hierarchy and select a class folder for the build path. The selection dialog also allows you to create a new folder.
Edit	Allows you to modify the currently selected library entry or entry attribute
Remove	Removes the selected element from the build path. This does not delete the resource.

Libraries have the following attributes (presented as library entry children nodes):

Library entry attributes

Attribute	Description
Javadoc location	

Projects tab

	Specifies where the library's Javadoc documentation can be found. If specified you can use Shift+F2 on an element of this library to open its documentation.
Source attachment	Specifies where the library's source can be found.

Order and Export tab

In the **Build class path order** list, you can click the **Up** and **Down** buttons to move the selected path entry up or down in the build path order for this new project.

Checked list entries are marked as exported. Exported entries are visible to projects that require the project. Use the **Select All** and **Deselect All** to change the checked state of all entries. Source folders are always exported, and can not be deselected.

Default output folder

At the bottom of this page, the **Default output folder** field allows you to enter a path to a folder path where the compilation output for this project will reside. The default output is used for source folders that do not specify an own output folder. Use **Browse** to select an existing location from the current project.

■ Related concepts

[Build classpath](#)
[Classpath variables](#)

■ Related tasks

[Working with build paths](#)
[Attaching source to variables](#)
[Attaching source to a JAR file](#)

■ Related reference

[Frequently asked questions on JDT](#)
[Classpath Variables preferences](#)
[Java Compiler properties](#)

File actions

File Menu Commands:

Name	Function	Keyboard Shortcut
New	Create a Java element or a new resource. Configure which elements are shown in the submenu in Window > Customize Perspective. In a Java perspective, by default action for creating a <u>project</u> , <u>package</u> , <u>class</u> , <u>enum</u> , <u>interface</u> , <u>annotation</u> , <u>source folder</u> , <u>scrapbook</u> , file and folder are available.	Ctrl + N
Close	Close the current editor. If the editor contains unsaved data, a save request dialog will be shown.	Ctrl + F4
Close All	Close all editors. If editors contains unsaved data, a save request dialog will be shown.	Ctrl + Shift + F4
Save	Save the content of the current editor. Disabled if the editor does not contain unsaved changes.	Ctrl + S
Save As	Save the content of the current editor under a new name.	
Save All	Save the content of all editors with unsaved changes. Disabled if no editor contains unsaved changes.	Ctrl + Shift + S
Revert	Revert the content of the current editor back to the content of the saved file. Disabled if the editor does not contain unsaved changes.	
Move	Move a resource. Disabled on Java Elements. To move Java elements use Refactor > Move (with updating all references to the file) or <u>Edit</u> > Cut / Paste (no updating of references).	
Rename	Renames a resource. Disabled on Java Elements. To rename Java elements use Refactor > Rename (with updating all references to the file).	
Refresh	Refreshes the content of the selected element with the local file system. When launched from no specific selection, this command refreshes all projects.	
Print	Prints the content of the current editor. Enabled when an editor has the focus.	Ctrl + P
Import	Opens the import wizard dialog. JDT does not contribute any import wizards.	
Export	Opens the export wizard dialog. JDT contributes the <u>JAR file export wizard</u> and the <u>Javadoc generation wizard</u> .	
Properties	Opens the property pages of the select elements. Opened on Java projects the <u>Java Build Path</u> page and the <u>Javadoc Location page</u> are available. For JAR archives, configure the JAR's <u>Source Attachment</u> and <u>Javadoc Location</u> here.	Alt + Enter
Exit	Exit Eclipse	

■ Related concepts

Java development tools (JDT)

■ **Related tasks**

Creating Java elements

Creating JAR Files

■ **Related reference**

New Java Project wizard

New Java Package wizard

New Java Class wizard

New Java Enum wizard

New Java Interface wizard

New Java Annotation wizard

New Java Scrapbook Page wizard

JAR file exporter

Javadoc generation

Javadoc Location properties

Java Build Path properties

Source Attachment properties

New Java Project Wizard

This wizard helps you create a new Java project in the workbench.

Project name page

Option	Description	Default
Project name	Type a name for the new project.	<blank>
Contents	<p>Create new project in workspace: When selected, the New Project Wizard will create a new project with the specified name in the workspace.</p> <p>Create project from existing source: When selected, you can specify the location from which the New Java Project Wizard will retrieve an existing Java project. Click on Browse... to browse for a location of an existing Java project.</p>	workspace
JDK Compliance	<p>Use default compiler compliance: When selected, the New Java Project Wizard creates a new Java project with the default compiler compliance. The default compiler compliance can be configured on the Compiler preference page.</p> <p>Use project specific compliance: When selected, you can explicitly specify the compiler compliance of the new Java project.</p>	default compliance
Project layout	<p>Use project folder as root for sources and class files: When selected, the project folder is used both as source folder and as output folder for class files.</p> <p>Create separate source and output folders: When selected, the New Java Project Wizard creates a source folder for Java source files and an output folder which holds the class files of the project.</p>	Use project folder

Java settings page

You can skip this page by pressing Finish on the first page. Otherwise press Next to configure the [Java Build Path](#).

If the project location on the first page has been set to an existing directory, you will be prompted if the New Java Project wizard should try to detect existing classpath settings. To do this the project will be created early on the invocation of the Next button, and the Back will be disabled on the Java Settings page.

Related concepts

[Java projects](#)

■ Related tasks

[Creating a new Java project](#)

■ Related reference

[File actions](#)

Java Compiler

This preference page lets you configure the various settings related to compiling, building and checking Java source code.

The Java compiler preferences are separated in the following sections:

- General
- Building
- Errors/Warnings

General

JDK Compliance

Option	Description	Default
Compiler compliance level	Specifies the compiler compliance level.	1.4
Use default compliance settings	If enabled, the default compliance settings for the compiler compliance level are applied.	On
Generated class files compatibility	Specifies the generated class file compatibility.	1.2
Source compatibility	Specifies the compatibility of the accepted source code.	1.3
Disallow identifiers called 'assert'	When enabled, the compiler will issue an error or a warning whenever 'assert' is used as an identifier (reserved keyword in J2SE 1.4).	Warning
Disallow identifiers called 'enum'	When enabled, the compiler will issue an error or a warning whenever 'enum' is used as an identifier (reserved keyword in J2SE 5.0).	Warning

Classfile generation

Add variable attributes to generated class files	If enabled, variable attributes are added to the class file. This will enable local variable names to be displayed in the debugger (in places where variables are definitely assigned) The resulting .class file is then bigger.	On
Add line number attributes to generated class files	If enabled, line number information is added to the class file. This will enable source code highlighting in the debugger.	On
Add source file name to generated class file		On

Basic tutorial

	If enabled, the source file name is added to the class file. This will enable the debugger to present the corresponding source code.	
Preserve unused local variables	If enabled, unused local variables (i.e. never read) are not stripped from the class file. If stripped this potentially alters debugging.	On
Inline finally blocks	If enabled, finally blocks are inlined in the generated class files. This positively affects performance, but may result in larger class files.	Off

Building

General

Option	Description	Default
Maximum number of reported problems per compilation unit	Specifies how many problems should be reported for a compilation unit.	100
Enable using exclusion patterns in source folders	When disabled, no entry on a project classpath can be associated with an exclusion pattern.	On
Enable using multiple output locations for source folders	When disabled, no entry on a project classpath can be associated with a specific output location, preventing thus usage of multiple output locations.	On

Build path problems

Abort building on build path errors	Allow to toggle the builder to abort if the classpath is invalid.	On
Incomplete build path	Indicate the severity of the problem reported when an entry on the classpath does not exist, is not legitimate or is not visible (e.g. a reference project is closed).	Error
Circular dependencies		Error

Basic tutorial

	Indicate the severity of the problem reported when a project is involved in a cycle.	
Incompatible required binaries	Indicated the severity of the problem reported when a project requires incompatible binaries.	Ignore

Output folder

Duplicated resources	Indicate the severity of the problem reported when more than one occurrence of a resource is to be copied into the output location.	Warning
Scrub output folders when cleaning projects	Indicate whether the Java Builder is allowed to clean the output folders when performing full build operations.	On
Filtered resources	A comma separated list of file patterns which are not copied to the output folder.	"

Errors/Warnings

Code style

Option	Description	Default
Non-static access to a static member	When enabled, the compiler will issue an error or a warning whenever a static field or method is accessed with an expression receiver. A reference to a static member should be qualified with a type name.	Warning
Indirect access to a static member	When enabled, the compiler will issue an error or a warning whenever a static field or method is indirectly accessed. A static field of an interface should be qualified with the declaring type name.	Warning
Unqualified access to instance field	When enabled, the compiler will issue an error or a warning whenever it encounters a field access which is not qualified (eg. misses a 'this').	Ignore
Undocumented empty block	When enabled, the compiler will issue an error or a warning whenever it encounters an empty block statement with no explaining comment.	Ignore
Access to a non-accessible	When enabled, the compiler will issue an error or a warning	Ignore

Basic tutorial

member of an enclosing type	whenever it emulates access to a non-accessible member of an enclosing type. Such accesses can have performance implications.	
Methods with a constructor name	Naming a method with a constructor name is generally considered poor style programming. When enabling this option, the compiler will signal such scenario either as an error or a warning.	Warning
Usage of non-externalized strings	When enabled, the compiler will issue an error or a warning for non externalized String literal (i.e. non tagged with <code>//NON-NLS-<n>\$</code>).	Ignore

Potential programming problems

Serializable class without serialVersionUID	When enabled, the compiler will issue an error or a warning whenever a type implementing 'java.io.Serializable' does not contain a serialVersionUID field.	Warning
Assignment has no effect (eg. 'x = x')	When enabled, the compiler will issue an error or a warning whenever an assignment has no effect (eg. 'x = x').	Warning
Possible accidental boolean assignm (eg. 'if (a = b)')	When enabled, the compiler will issue an error or a warning whenever it encounters a possible accidental boolean assignment (eg. 'if (a = b)').	Warning
'finally' does not complete normally	When enabled, the compiler will issue an error or a warning whenever a 'finally' statement does not complete normally (eg. contains a return statement).	Warning
Empty statement	When enabled, the compiler will issue an error or a warning whenever it encounters an empty statement (eg. a superfluous semicolon).	Ignore
Using a char array in string concatenation	When enabled, the compiler will issue an error or a warning whenever a char[] expression is used in String concatenations, <code>"hello" + new char[]{'w','o','r','l','d'}</code>	Warning
Hidden catch blocks	Locally to a try statement, some catch blocks may hide others , eg. <pre>try { throw new java.io.CharConversionException(); } catch (java.io.CharConversionException e) { } catch (java.io.IOException e) {}.</pre> When enabling this option, the compiler will issue an error or a warning for hidden catch blocks corresponding to checked exceptions.	Warning

Name shadowing and conflicts

Field declaration hides another field or	When enabling this option, the compiler will issue an error or a warning if a field declaration hides another inherited field.	Ignore
--	--	--------

Basic tutorial

variable		
Local variable declaration hides another field or variable	When enabling this option, the compiler will issue an error or a warning if a local variable declaration hides another field or variable.	Ignore
Include constructor or setter method parameters	When enabling this option, the compiler additionally will issue an error or a warning if a constructor or setter method parameter hides another field or variable.	Off
Type parameter hides another type	When enabling this option, the compiler will issue an error or a warning if eg. a type parameter of an inner class hides an outer type.	Warning
Methods overridden but not package visible	A package default method is not visible in a different package, and thus cannot be overridden. When enabling this option, the compiler will signal such scenario either as an error or a warning.	Warning
Conflict of interface method with protected 'Object' method	<p>When enabled, the compiler will issue an error or a warning whenever an interface defines a method incompatible with a non-inherited Object method. Until this conflict is resolved, such an interface cannot be implemented, eg.</p> <pre>interface I { int clone(); }</pre>	Warning

Name shadowing and conflicts

Deprecated API	When enabled, the compiler will signal use of deprecated API either as an error or a warning.	Warning
Signal use of deprecated API inside deprecated code	When enabled, the compiler will signal use of deprecated API inside deprecated code. The severity of the problem is controlled with option "Deprecated API".	Off
Signal overriding or implementing deprecated method	When enabled, the compiler will signal overriding or implementing a deprecated method The severity of the problem is controlled with option "Deprecated API".	Off
Forbidden reference (access rules)	When enabled, the compiler will signal a forbidden reference specified in the access rules.	Error
Discouraged reference (access rules)	When enabled, the compiler will signal a discouraged reference specified in the access rules.	Warning

Unnecessary code

Local variable is never read	When enabled, the compiler will issue an error or a warning whenever a local variable is declared but never used within the its	Warning
------------------------------	---	---------

Basic tutorial

	scope.	
Parameter is never read	When enabled, the compiler will issue an error or a warning whenever a parameter is declared but never used within the its scope.	Ignore
Check overriding and implementing methods	When enabled, the compiler additionally will issue an error or a warning whenever a parameter is declared but never used within the its scope in overriding or implementing methods.	Off
Unused imports	When enabled, the compiler will issue an error or a warning for unused import reference.	Warning
Unused local or private members	When enabled, the compiler will issue an error or a warning whenever a local or private member is declared but never used within the same unit.	Warning
Unnecessary else statement	When enabled, the compiler will issue an error or a warning whenever it encounters an unnecessary else statement (eg. if (condition) return; else doSomething();).	Ignore
Unnecessary cast or 'instanceof' operation	When enabled, the compiler will issue an error or a warning whenever it encounters an unnecessary cast or 'instanceof' operation (eg. if (object instanceof Object) return;).	Ignore
Unnecessary declaration of thrown checked exception	When enabled, the compiler will issue an error or a warning whenever it encounters an unnecessary declaration of a thrown exception.	Ignore
Check overriding and implementing methods	When enabled, the compiler additionally will issue an error or a warning whenever it encounters an unnecessary declaration of a thrown exception in an overriding or implementing method.	Off

J2SE 5.0 options

Unchecked generic type operation	When enabled, the compiler will issue an error or a warning whenever it encounters an unchecked generic type operation.	Warning
Generic type parameter declared with a final type bound	When enabled, the compiler will issue an error or a warning whenever it encounters a type bound involving a final type.	Warning
Inexact type match for vararg arguments	When enabled, the compiler will issue an error or a warning whenever it encounters an inexact type match for vararg arguments.	Warning
Boxing and unboxing conversions	When enabled, the compiler will issue an error or a warning whenever it encounters a boxing or	Ignore

Basic tutorial

	unboxing conversion. Autoboxing may affects performance negatively.	
Missing '@Override' annotation	When enabled, the compiler will issue an error or a warning whenever it encounters a method overriding another implemented method, and the '@Override' annotation is missing.	Ignore
Missing '@Deprecated' annotation	When enabled, the compiler will issue an error or a warning whenever it encounters a deprecated type without additional '@Deprecated' annotation.	Ignore
Annotation is used as super interface	When enabled, the compiler will issue an error or a warning whenever it encounters a type implementing an annotation. Although possible, this is considered bad practice.	Warning
Not all enum constants covered on 'switch'	When enabled, the compiler will issue an error or a warning whenever it encounters a switch statement which does not contain case statements for every enum constant of the referenced enum.	Ignore
Unhandled warning tokens in '@SuppressWarnings'	When enabled, the compiler will issue an error or a warning whenever it encounters an unhandled warning token in a '@SuppressWarnings' annotation.	Warning
Enable '@SuppressWarnings' annotations	When enabled, the compiler will process '@SuppressWarnings' annotations.	On

■ Related concepts

Java builder

■ Related tasks

Building a Java program

Working with build paths

Working with JREs

■ Related reference

Classpath Variables preferences

Java Build Path properties

Building a Java program

A build command compiles workbench resources. A build command can be triggered in different ways:

- Building automatically: If auto build is turned on (***Project > Build Automatically***, or ***Window > Preferences > General > Workspace > Build automatically***), then an incremental build occurs every time you save a modified workbench resource.
- Building manually: You can perform a manual build using a keyboard shortcut, a project's pop-up menu or the ***Project*** menu in the menu bar.

■ Related concepts

[Build classpath](#)

[Java builder](#)

■ Related tasks

[Building automatically](#)

[Building manually](#)

[Adding a JAR file to the build path](#)

[Adding a library folder to the build path](#)

[Viewing and editing a project's build path](#)

[Working with build paths](#)

■ Related reference

[Java Build Path](#)

[Java Compiler preferences](#) [Project menu](#)

Building automatically

To enable automatic building:

Enable *Project > Build Automatically* or
select the *Window > Preferences > General > Workspace > Build automatically* checkbox.

To disable automatic building:

Disable *Project > Build Automatically* or
clear the *Window > Preferences > General > Workspace > Build automatically* checkbox.

■ Related concepts

[Java builder](#)

[Build class path](#)

■ Related tasks

[Building a Java program](#)

[Building manually](#)

[Viewing compilation errors and warnings](#)

[Working with build paths](#)

[Adding a JAR file to the build path](#)

[Adding a library folder to the build path](#)

[Viewing and editing a project's build path](#)

■ Related reference

[Java Build path](#)

Building manually

Incremental build

The **Build** command performs an incremental build that compiles all resources modified since the last build. To trigger an incremental build, either:

- press **Ctrl+B** or
- from the menu bar, select **Project > Build All**

Incremental project build

You can also incrementally build single projects. Select the project that you want to build and:

- Select **Project > Build Project** from the menu bar or
- Select **Build Project** from the project's pop-up menu

Clean and rebuild from scratch (full build)

To clean and rebuild all workbench resources that have an associated builder (e.g. Java projects), select **Project > Clean...** from the menu bar. Then select **Clean all projects** and press **OK**.

Clean and rebuild selected projects

To clean and rebuild all resources contained in one or multiple projects:

- Select the projects
- Select **Project > Clean...** from the menu bar
- Press **OK**

or

- Select **Project > Clean...** from the menu bar
- Select **Clean projects selected below**
- Select the projects to clean and press **OK**

Related concepts

[Java builder](#)

[Build classpath](#)

Related tasks

[Building a Java program](#)

[Building automatically](#)

[Working with build paths](#)

[Adding a JAR file to the build path](#)

[Adding a library folder to the build path](#)
[Viewing and editing a project's build path](#)

■ Related reference

[Java Build path](#)
[Project menu](#)

Working with build paths

Setting up the proper Java build path is an important task when doing Java development. Without the correct Java build path, you cannot compile your code. In addition, you cannot search or look at the type hierarchies for Java elements.

■ Related concepts

[Java builder](#)

[Build classpath](#)

■ Related tasks

[Viewing and editing a project's build path](#)

[Adding a JAR file to the build path](#)

[Adding a library folder to the build path](#)

[Adding source code as individual files](#)

[Assigning the default JRE for the workbench](#)

[Building a Java program](#)

[Building automatically](#)

[Building manually](#)

[Choosing a JRE for launching a project](#)

■ Related reference

[Java Build path](#)

[Installed JREs preference page](#)

Viewing and editing a project's Java build path

A project's Java build path can either be defined when creating the project with the *New Wizard* or later in the project's property dialog. The Java build path settings dialog is the same in both cases. To view and edit a project's Java build path, follow these steps:

1. Select the project you want to view or edit
2. From the project's pop-up menu, select *Properties*
3. Select the *Java Build Path* page
4. Define the source entries for the build path on the *Source* page:
 - ◆ Click the *Add Folder* button to add source folders to the Java build path.
The *Remove* button removes the selected folder(s) from the build path.
Edit lets you modify the selected entry.
5. On the *Projects* page, identify the other projects that are required for building this project. The list shows all the existing Java projects from the workbench. *Note*: Each selected project is automatically added to the list of referenced projects.
6. On the *Libraries* page, define the libraries required by your project. Libraries come in different forms. There are buttons for adding a library in each form. By default, each Java project has a 'JRE System Library' entry on the build path. This entry stands for the workbench's default JRE.
7. On the *Order and Export* page, define the Java build path order. The recommended ordering is to have source entries before the library entries and the required projects.

■ Related concepts

[Java builder](#)

[Build classpath](#)

■ Related tasks

[Adding a JAR file to the build path](#)

[Adding a library folder to the build path](#)

[Adding a variable class path entry](#)

[Building a Java program](#)

[Working with build paths](#)

[Working with JREs](#)

■ Related reference

[Java Build path](#)

Adding a JAR file to the build path

You can add a JAR file stored either in the workbench or anywhere in your file system to the build class path.

To add a JAR to your build class path follow these steps:

1. Select the project, and from its pop-up menu, select **Properties**. In the Properties dialog, select the **Java Build Path** page.
Click the **Libraries** tab.
You can now either add a JAR file which is contained in your workspace or which is somewhere else:
 - ◆ to add a JAR file which is inside your workspace click the **Add JARs** button
 - ◆ to add an external JAR file click the **Add External JARs** button
2. In the dialog that appears, select the JAR file that you want to add. Note that you can add multiple JARs at once.

Alternatively, to add an external JAR to your build class path follow these steps:

1. Select the project, and from its pop-up menu, select **Build Path > Add external archives**.
2. In the dialog that appears, select the JAR file that you want to add. Note that you can add multiple JARs at once.

■ Related concepts

[Java builder](#)

[Build classpath](#)

■ Related tasks

[Adding a library folder to the build path](#)

[Building a Java program](#)

[Building automatically](#)

[Building manually](#)

[Creating a new JAR file](#)

[Viewing and editing a project's build path](#)

[Working with build paths](#)

■ Related reference

[Java Build path](#)

Adding a library folder to the build path

A library folder is an ordinary folder containing a collection of class files inside the workbench. Use this format for a library when a library is not packaged as a JAR file.

To add a library folder to the project's build class path, follow these steps:

1. Select the project, and from its context menu, select **Properties**.
2. In the Properties dialog, select the **Java Build Path** page.
3. Click the **Libraries** tab.
4. Click the **Add Class Folder** button.
5. In the dialog that appears, select a folder to add press the **OK** button. If you want to add a not yet existing folder use first **Create New Folder**. After the folder is created select the new folder and press the **OK** button

■ Related concepts

[Java builder](#)

[Build classpath](#)

■ Related tasks

[Adding a JAR file to the build path](#)

[Building a Java program](#)

[Building automatically](#)

[Building manually](#)

[Viewing and editing a project's build path](#)

[Working with build paths](#)

■ Related reference

[Java Build Path](#)

Creating a new JAR file

To create a new JAR file in the workbench:

1. In the Package Explorer, you can optionally pre-select one or more Java elements to export. (These will be automatically selected in the **JAR Package Specification** wizard page, described in Step 4.)
2. Either from the context menu or from the menu bar's **File** menu, select **Export**.
3. Select **JAR file**, then click **Next**.
4. In the **JAR Package Specification** page, select the resources that you want to export in the **Select the resources to export** field.
5. Select the appropriate checkbox to specify whether you want to **Export generated class files and resources** or **Export Java source files and resources**. **Note:** Selected resources are exported in both cases.
6. In the **Select the export destination** field, either type or click **Browse** to select a location for the JAR file.
7. Select or clear the **Compress the contents of the JAR file** checkbox.
8. Select or clear the **Overwrite existing files without warning** checkbox. If you clear this checkbox, then you will be prompted to confirm the replacement of each file that will be overwritten.
9. **Note:** The overwrite option is applied when writing the JAR file, the JAR description, and the manifest file.
10. You have two options:
 - ◆ Click **Finish** to create the JAR file immediately.
 - ◆ Click **Next** to use the JAR Packaging Options page to set advanced options, create a JAR description, or change the default manifest.

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Adding a JAR file to the build path](#)

[Attaching source to a JAR file](#)

[Defining the JAR file's manifest](#)

[Setting advanced options](#)

■ Related reference

[JAR file exporter](#)

[Package Explorer](#)

Attaching source to a JAR file

You can attach source to a JAR file to enable source-level stepping and browsing of classes contained in a binary JAR file. Unless its source code is attached to a JAR file in the workbench, you will not be able to view the source for the JAR file.

To attach source to a JAR file:

1. Select the project, and from its pop-up menu, select **Properties**.
In the Properties dialog, select the Java Build Path page.
2. On the **Libraries** tab, select the JAR file to which you want to attach source.
Expand the node by clicking on the plus and select the node *Source Attachment*. Click the **Edit** button to bring up the source attachment dialog.
3. Fill in the **Location path** field depending on the location, choose between the workspace, an external file or external folder.
4. Click **OK**.

or

1. Select the JAR file in the Package Explorer, and from its pop-up menu, select **Properties**.
In the Properties dialog, select the **Java Source Attachment** page.
2. Fill in the **Location path** field depending on the location, choose between the workspace, an external file or external folder.
3. Click **OK**.

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Attaching source to variables](#)

[Creating a new JAR file](#)

[Stepping through the execution of a program](#)

■ Related reference

[Java Build Path](#)

[Source Attachment dialog](#)

Attaching source to a class path variable

When attaching source for a class path variable entry, both the path to the *Archive* and the *Root Path* must be defined by variables. Follow these steps to make a source attachment for a variable:

1. Select the project, and from its pop-up menu, select **Properties**.
In the Properties dialog, select the Java Build Path page.
2. On the **Libraries** tab, select the class path variable to which you want to attach source.
Expand the node by clicking on the plus and select the node *Source Attachment*. Click the **Edit** button to bring up the source attachment dialog.
3. In **Location variable path** field, use a variable to define the location of the archive. Use the **Variable** button to select an existing variable and the **Extension** button to append an optional path extension.
4. Click **OK**.

or

1. Select the JAR file in the Package Explorer, and from its pop-up menu, select **Properties**.
In the Properties dialog, select the **Java Source Attachment** page.
2. In **Location variable path** field, use a variable to define the location of the archive. Use the **Variable** button to select an existing variable and the **Extension** button to append an optional path extension.
3. Click **OK**.

■ Related concepts

[Classpath variables](#)

■ Related tasks

[Adding a variable class path entry](#)

[Defining a class path variable](#)

[Deleting a class path variable](#)

■ Related reference

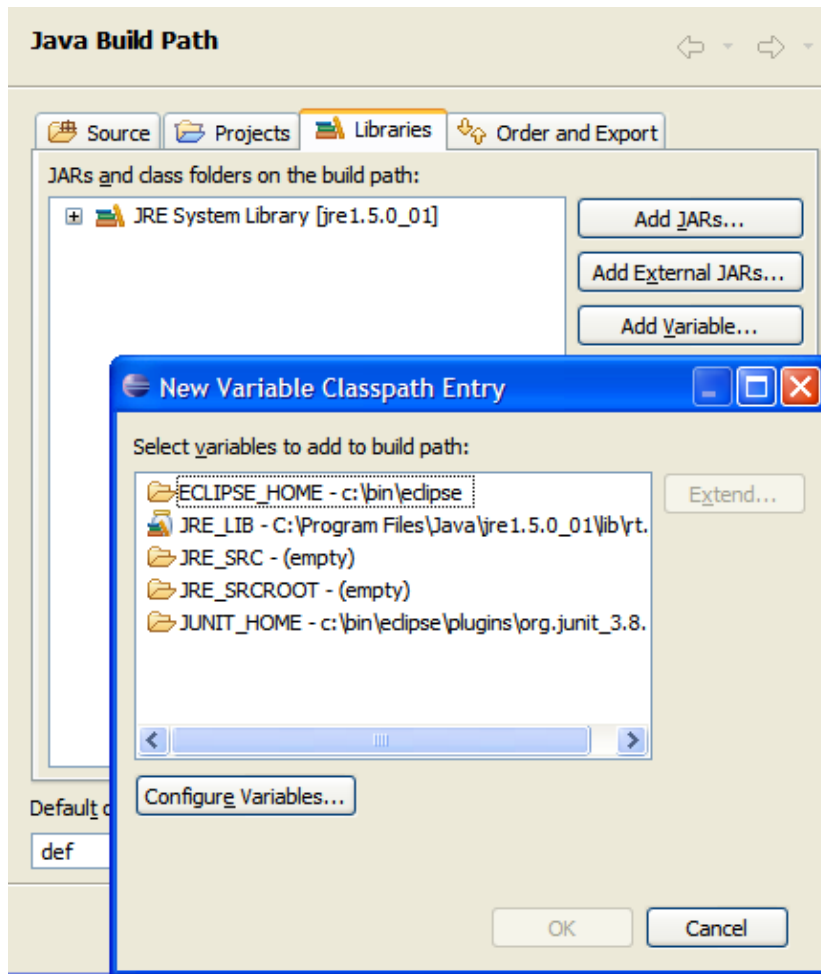
[Classpath Variables preference page](#)

[Source Attachment dialog](#)

Adding a classpath variable to the build path

To add a classpath variable to the Java build path of a project, follow these steps:

1. Select the project to which you want to add the classpath variable
2. From the project's pop-up menu, select **Properties**
3. In the Properties page, select the **Java Build Path** page.
4. On the **Libraries** tab, click **Add Variable** for adding a variable that refers to a JAR file.
The **New Variable Classpath Entry** dialog appears which shows all available classpath variables.



5. Select a classpath variable then press **OK**.
 - ◆ If the variable resolves to a folder, you can specify a path extension that points to a JAR. To do this press the **Extend...** button.
 - ◆ Press **Edit...** to create a new classpath variable or edit an existing one.

Hint: You can add multiple variable entries at once to the Java build path: Select more than one variable in the **New Variable Classpath Entry** dialog, or multiple JAR files in the **Variable Extension** dialog.

■ Related concepts

[Classpath variables](#)

[Build classpath](#)

■ Related tasks

[Attaching source to a class path variable](#)

[Creating Java elements](#)

[Defining a class path variable](#)

[Deleting a class path variable](#)

[Viewing and editing a project's build path](#)

■ Related reference

[Classpath Variables preference page](#)

[Build path properties page](#)

Defining a classpath variable

Classpath variables are stored global to the workbench; in other words, all projects in the workbench can share the classpath variables.

To add or change a class path variable, follow these steps:

1. From the menu bar, select **Window > Preferences**.
Select the **Java > Build Path > Classpath Variables** page.
2. This page allows you to add, edit, or remove class path variables.
 - ◆ To add a new class path variable, click the **New...** button. The New Variable Entry page opens.
 - ◆ To edit an existing class path variable, select the variable in the **Defined class path variables** list and click the **Edit...** button. The Edit Variable Entry page opens. *Note: The reserved class path variables, JRE_LIB, JRE_SRC, and JRE_SRCROOT cannot be edited in this page. To change them, change the default workbench JRE on the Installed JREs page (Window > Preferences > Java > Installed JREs).*Type a name for the variable in the **Name** field.
3. Type a path to be referenced by the variable in the **Path** field. You can also click the **File** or **Folder** buttons to browse the file system.
4. Click **OK** when you are done. The new or modified variable appears in the **Defined class path variables** list on the Preferences page.

■ Related concepts

Classpath variables

■ Related tasks

Adding a variable class path entry

Attaching source to a class path variable

Creating Java elements

Deleting a class path variable

■ Related reference

Classpath Variables preference page

Deleting a classpath variable

To delete an existing class path variable:

1. From the menu bar, select *Window > Preferences*.
2. Select the *Java > Build Path > Classpath Variables* page.
3. Select the variable(s) you want to delete in the *Defined class path variables* list and click the *Remove* button. The variable(s) are removed from the classpath variables list on the preferences page.

Note: The reserved class path variables, JRE_LIB, JRE_SRC, and JRE_SRCROOT cannot be deleted.

■ Related concepts

[Classpath variables](#)

■ Related tasks

[Adding a variable class path entry](#)

[Attaching source to a class path variable](#)

[Creating Java elements](#)

[Defining a class path variable](#)

■ Related reference

[Classpath Variables preference page](#)

Classpath variables

Configurable variables

Classpath variables can be used in a Java Build Path to avoid a reference to the local file system. Using a variable entry, the classpath only contains a variable and the build path can be shared in a team. The value of the variable has to be configured on this page.

Command	Description
<i>New...</i>	Adds a new variable entry. In the resulting dialog, specify a name and path for the new variable. You can click the File or Folder buttons to browse for a path.
<i>Edit...</i>	Allows you to edit the selected variable entry. In the resulting dialog, edit the name and/or path for the variable. You can click the File or Folder buttons to browse for a path.
<i>Remove</i>	Removes the selected variable entry.

Reserved class path variables

Certain class path variables are set internally and can not be changed in the Classpath variables preferences:

- JRE_LIB: The archive with the runtime JAR file for the currently used JRE.
- JRE_SRC: The source archive for the currently used JRE.
- JRE_SRCROOT: The root path in the source archive for the currently used JRE.

■ Related concepts

[Classpath variables](#)

■ Related tasks

[Working with JREs](#)

[Working with build paths](#)

[Adding a variable classpath entry](#)

[Attaching source to a classpath variable](#)

[Defining a classpath variable](#)

[Deleting a classpath variable](#)

■ Related reference

[Installed JREs](#)

Working with JREs

You can install as many different Java Runtime Environments (JREs) as you like. A JRE definition consists of:

- The type of the JRE (e.g. Standard VM or Standard 1.x.x VM)
- A name
- The location where the JRE is installed
- The location (URL) of the Javadoc
- The system libraries containing the Java system classes (like java.lang.Object). Optionally, the system libraries can be associated with the source file containing the source for the classes in the JRE's CLASS files

You can switch the default JRE for the workbench. The default JRE is the JRE to which the pre-defined classpath variables JRE_LIB, JRE_SRC and JRE_SRCROOT are bound.

■ Related concepts

[Java development tools \(JDT\)](#)

[Classpath variable](#)

■ Related tasks

[Adding a new JRE definition](#)

[Assigning the default JRE for the workbench](#)

[Choosing a JRE for launching a project](#)

[Deleting a JRE definition](#)

[Editing a JRE definition](#)

[Overriding the default system libraries for a JRE definition](#)

[Viewing and editing a project's build path](#)

■ Related reference

[Installed JREs preference page](#)

Adding a new JRE definition

You can add any number of JRE definitions.

1. From the menu bar, select **Window > Preferences**.
2. In the left pane, expand the **Java** category and select **Installed JREs**.
3. Click the **Add...** button. The Create JRE dialog opens.
4. In the **JRE type** field, select the type of JRE you want to add from the drop-down list.
5. In the **JRE name** field, type a name for the new JRE definition. All JREs of the same type must have a unique name.
6. In the **JRE home directory** field, type or click **Browse** to select the path to the root directory of the JRE installation (usually the directory containing the *bin* and *lib* directories for the JRE). This location is checked automatically to make sure it is a valid path.
7. In the **Javadoc URL** field, type or click **Browse** to select the URL location. The location is used by the Javadoc export wizard as a default value and by the 'Open External Javadoc' action.
8. If you want to use the default libraries and source files for this JRE, select the **Use default system libraries** checkbox. Otherwise, clear it and customize as desired. Source can be attached for the referenced JARs as well.
9. Click **OK** when you are done.

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Working with JREs](#)

[Assigning the default JRE for the workbench](#)

[Editing a JRE definition](#)

[Overriding the default system libraries for a JRE definition](#)

■ Related reference

[Installed JREs preference page](#)

Assigning the default JRE for the workbench

The default JRE is used for compiling and launching Java programs in all projects unless you specifically override the default JRE. The default JRE is the installed JRE to which JRE_LIB, JRE_SRC and JRE_SRCROOT are bound. A project is not compiled against the default JRE if the JRE_LIB variable has been removed from its build path. A program is not launched with the default JRE if a custom runtime JRE has been set for its project.

Here is how you can change the default JRE:

1. From the workbench's menu bar, select **Window > Preferences**.
2. Expand the **Java** category in the left pane and select **Installed JREs**.
3. Check the box on the line for the JRE that you want to assign as the default JRE in your workbench. If the JRE you want to assign as the default does not appear in the list, you must add it.
4. Click **OK**.

Note: The JRE_LIB, JRE_SRC and JRE_SRCROOT system variables are automatically updated when you change the default JRE. This may cause a build to occur if you have auto build enabled (**Project > Build Automatically** or **Window > Preferences > General > Workspace > Build automatically**).

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Adding a new JRE definition](#)

[Choosing a JRE for launching a project](#)

[Working with build paths](#)

[Working with JREs](#)

■ Related reference

[Installed JREs preference page](#)

Choosing a JRE for a launch configuration

Instead of using the default JRE for running and debugging all Java Application launch configurations, you can specifically assign a JRE for launching an individual configuration.

1. With a Java Application configuration selected in the Launch Configuration Dialog, select the JRE tab.
2. In the list of available JREs, select the JRE you want to use to launch this configuration and click ***Apply***, ***Run***, or ***Debug***.

Note: Changing the JRE used for running does not affect the way Java source is compiled. You can adjust the build path to compile against custom libraries.

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Assigning the default JRE for the workbench](#)

[Running and debugging](#)

[Working with build paths](#)

[Working with JREs](#)

Running and debugging

You may launch your Java programs from the workbench. The programs may be launched in either run or debug mode.

- In run mode, the program executes, but the execution may not be suspended or examined.
- In debug mode, execution may be suspended and resumed, variables may be inspected, and expressions may be evaluated.

■ Related concepts

[Debugger](#)

[Remote debugging](#)

[Local debugging](#)

■ Related tasks

[Changing debugger launch options](#)

[Choosing a JRE for launching a project](#)

[Creating a Java scrapbook page](#)

[Disconnecting from a VM](#)

[Launching a Java program](#)

[Local debugging](#)

[Preparing to debug](#)

[Re-launching a program](#)

[Remote debugging](#)

[Resuming the execution of suspended threads](#)

[Setting execution arguments](#)

[Stepping through the execution of a program](#)

[Suspending threads](#)

[Viewing compilation errors and warnings](#)

■ Related reference

[Run and debug actions](#)

[Debug view](#)

[Debug preferences](#)

[Console preferences](#)

Remote debugging

The client/server design of the Java debugger allows you to launch a Java program from computer on your network and debug it from the workstation running the platform. This is particularly useful when you are developing a program for a device that cannot host the development platform. It is also useful when debugging programs on dedicated machines such as web servers.

Note: To use remote debugging, you must be using a Java VM that supports this feature.

To debug a program remotely, you must be able to launch the program in debug mode on the remote machine so that it will wait for a connection from your debugger. The specific technique for launching the program and connecting the debugger are VM-specific. The basic steps are as follows:

1. Ensure that you are building your Java program with available debug information. (You can control these attributes from **Window > Preferences > Java > Compiler**).
2. After you build your Java program, install it to the target computer. This involves copying the .CLASS files or .JAR files to the appropriate location on the remote computer.
3. Invoke the Java program on the remote computer using the appropriate VM arguments to specify debug mode and a communication port for the debugger.
4. Start the debugger using a remote launch configuration and specify the address and port of the remote computer.

More specific instructions for setting up a launch configuration for remote debugging should be obtained from your VM provider.

■ Related tasks

[Using the remote Java application launch configuration](#)
[Disconnecting from a VM](#)

Using the remote Java application launch configuration

The **Remote Java Application** launch configuration should be used when debugging an application that is running on a remote VM. Since the application is started on the remote system, the launch configuration does not specify the usual information about the JRE, program arguments, or VM arguments. Instead, information about connecting to the application is supplied.

To create a **Remote Java Application** launch configuration, do the following:

1. Select **Run >Debug...** from the workbench menu bar (or **Debug...** from the drop-down menu on the **Debug** tool bar button) to show the launch configuration dialog.
2. Select the **Remote Java Application** in the list of configuration types on the left.
3. Click the **New** button. A new remote launch configuration is created and three tabs are shown: **Connect**, **Source**, and **Common**.
4. In the **Project** field of the **Connect** tab, type or browse to select the project to use as a reference for the launch (for source lookup). A project does not need to be specified.
5. In the **Host** field of the **Connect** tab, type the IP address or domain name of the host where the Java program is running.
If the program is running on the same machine as the workbench, type *localhost*.
6. In the **Port** field of the **Connect** tab, type the port where the remote VM is accepting connections. Generally, this port is specified when the remote VM is launched.
7. The **Allow termination of remote VM** flag is a toggle that determines whether the **Terminate** command is enabled in the debugger. Select this option if you want to be able to terminate the VM to which you are connecting.
8. Click **Debug**. The launch attempts to connect to a VM at the specified address and port, and the result is displayed in the Debug view. If the launcher is unable to connect to a VM at the specified address, an error message appears.

Specific instructions for setting up the remote VM should be obtained from your VM provider.

■ Related concepts

[Debugger](#)

■ Related tasks

[Launching a Java program](#)

[Disconnecting from a VM](#)

[Setting execution arguments](#)

■ Related reference

[Debug view](#)

Disconnecting from a VM

To disconnect from a VM that was connected to with a Remote Java Application launch configuration:

1. In the Debug view, select the launch.
2. Click the ***Disconnect*** button in the view's toolbar. Communication with the VM is terminated, and all threads in the remote VM are resumed. Although the remote VM continues to execute, the debug session is now terminated.

■ Related concepts

Debugger

■ Related tasks

Connecting to a remote VM with the Remote Java application launch configuration
Running and debugging

■ Related reference














Debug view

Debug view



This view allows you to manage the debugging or running of a program in the workbench. It displays the stack frame for the suspended threads for each target you are debugging. Each thread in your program appears as a node in the tree. It displays the process for each target you are running.

If the thread is suspended, its stack frames are shown as child elements.

Debug View Commands

Command	Name	Description
	Resume	This command resumes a suspended thread.
	Suspend	This command suspends the selected thread of a target so that you can browse or modify code, inspect data, step, and so on.
	Terminate	This command terminates the selected debug target.
 Context menu only	Terminate & Remove	This command terminates the selected debug target and removes it from the view.
 Context menu only	Terminate All	This command terminates all active launches in the view.
	Disconnect	This command disconnects the debugger from the selected debug target when debugging remotely.
	Remove All Terminated Launches	This command clears all terminated debug targets from the view display.
	Use Step Filters	This command toggles step filters on/off. When on, all step functions apply step filters.
	Step Into	This command steps into the highlighted statement.
	Step Over	<p>This command steps over the highlighted statement. Execution will continue at the next line either in the same method or (if you are at the end of a method) it will continue in the method from which the current method was called.</p> <p>The cursor jumps to the declaration of the method and selects this line.</p>
	Run to Return	This command steps out of the current method. This option stops execution after exiting the current method.
	Show Qualified Names	This option can be toggled to display or hide qualified names.
 Context	Copy Stack	This command copies the selected stack of suspended threads as well as the state of the running threads to the clipboard.

Basic tutorial

menu only		
	Drop to Frame	<p>This command lets you drop back and reenter a specified stack frame. This feature is similar to "running backwards" and restarting your program part-way through.</p> <p>To drop back and reenter a specified stack frame, select the stack frame that you want to "drop" to, and select Drop to Frame.</p> <p>Some caveats apply to this feature:</p> <ul style="list-style-type: none">• You cannot drop past a native method on the stack.• Global data are unaffected and will retain their current values. For example, a static vector containing elements will not be cleared. <p><i>Note: This command is only enabled if the underlying VM supports this feature.</i></p>
 Context menu only	Relaunch	This command re-launches the selected debug target.
Context menu only	Properties	This command displays the properties of the selected launch. It also allows you to view the full command line for a selected process.

■ Related concepts

[Debugger](#)

[Java views](#)

[Local debugging](#)

[Remote debugging](#)

■ Related tasks

[Changing debugger launch options](#)

[Connecting to a remote VM with the Remote Java application launch configuration](#)

[Disconnecting from a VM](#)

[Launching a Java program](#)

[Preparing to debug](#)

[Resuming the execution of suspended threads](#)

[Running and debugging](#)

[Stepping through the execution of a program](#)

[Suspending threads](#)

■ Related reference

[Debug preferences](#)

[Run and debug actions](#)

[Views and editors](#)

Debug view

Local debugging

The Java debugger has a client/server design so that it can be used to debug programs that run locally (on the same workstation as the debugger) or remotely (on another computer on the network).

Local debugging is the simplest and most common kind of debugging. After you have finished editing and building your Java program, you can launch the program on your workstation using the **Run > Debug...** menu item on the workbench. Launching the program in this way will establish a connection between the debugger client and the Java program that you are launching. You may then use breakpoints, stepping, or expression evaluation to debug your program.

■ Related concepts

Breakpoints

■ Related tasks

Adding breakpoints

Resuming the execution of suspended threads

Running and debugging

Suspending threads

■ Related reference

Debug preferences

Debug view

Run and debug actions

Resuming the execution of suspended threads

To resume the execution of a suspended threads:

1. Select the thread or its stack frame in the Debug view.
2. Click the **Resume** button in the Debug view toolbar (or press the **F8** key). The thread resumes its execution, and stack frames are no longer displayed for the thread. The Variables view is cleared.

■ Related concepts

Debugger

■ Related tasks

Evaluating expressions

Stepping through the execution of a program

Suspending threads

■ Related reference

Debug view

Evaluating expressions

When the VM suspends a thread (due to hitting a breakpoint or stepping through code), you can evaluate expressions in the context of a stack frame.

1. Select the stack frame in which an evaluation is to be performed. For the detail panes of the *Variables* and *Expressions* views, the evaluation context will be a selected variable. If no variable is selected, the selected stack frame will be the context.
2. Expressions can be entered and evaluated in the following areas:
 - ◆ *Display* view
 - ◆ Detail pane of the *Expressions* view
 - ◆ Detail pane of the *Variables* view
 - ◆ Java editor when it is displaying source and it is not read-only
3. Select the expression to be evaluated and select *Display*, *Inspect* or *Execute* from the context pop-up menu. The result of a *Display* or *Inspect* evaluation is shown in a popup window. Note that *Execute* does not display a result – the expression is just executed.
4. The result popup window can be dismissed by clicking outside of the popup window or by pressing *Esc*. The result can be persisted to the Display view (if *Display* was chosen) or Expressions view (if *Inspect* was chosen) by pressing the key sequence shown at the bottom of the popup window. For example, to move the result of an *Inspect* evaluation to the Expressions view press *CTRL-Shift-I*. Note that when the *Display* action is used from the Display view the result is written to the Display view rather than a popup

Note: Evaluations cannot be performed in threads that have been manually suspended.

■ Related concepts

[Debugger](#)

[Java editor](#)

■ Related tasks

[Suspending threads](#)

[Resuming the execution of suspended threads](#)

■ Related reference

[Display view](#)

[Expressions view](#)

[Expressions view Show Detail Pane](#)

[Variables view](#)

[Variables view Show Detail Pane](#)

Suspending threads

To suspend an executing thread:

1. Select the thread in the Debug view.
2. Click the ***Suspend*** button in the Debug view toolbar. The thread suspends its execution. The current call stack for the thread is displayed, and the current line of execution is highlighted in the editor in the Debug perspective.

When a thread suspends, the top stack frame of the thread is automatically selected. The Variables view shows the stack frame's variables and their values. Complex variables can be further examined by expanding them to show the values of their members.

When a thread is suspended and the cursor is hovered over a variable in the Java editor, the value of that variable is displayed.

■ Related concepts

[Debugger](#)

[Java editor](#)

[Java perspectives](#)

■ Related tasks

[Catching Java exceptions](#)

[Evaluating expressions](#)

[Resuming the execution of suspended threads](#)

[Stepping through the execution of a program](#)

■ Related reference

[Debug view](#)

[Variables view](#)

Catching Java exceptions

It is possible to suspend the execution of thread when an exception is thrown by specifying an exception breakpoint. Execution can be suspended at locations where the exception is uncaught, caught, or both.

1. Choose **Add Java Exception Breakpoint** from the Breakpoints view or the workbench Run menu.
2. A dialog listing all of the available exceptions is shown.
3. Either type the name of the exception you want to catch or select it from the list.
4. At the bottom of the page, use the checkboxes to specify how you want execution to suspend at locations where the exception is thrown.
 - ◆ Select **Caught** if you want execution to suspend at locations where the exception is thrown but caught.
 - ◆ Select **Uncaught** if you want execution to suspend at locations where the exception is uncaught.

Note: Exception breakpoints can be enabled and disabled and have hit counts just like regular breakpoints.

■ Related concepts

[Java development tools \(JDT\)](#)
[Breakpoints](#)

■ Related tasks

[Suspending threads](#)
[Adding breakpoints](#)
[Removing breakpoints](#)
[Enabling and disabling breakpoints](#)
[Setting method breakpoints](#)

■ Related reference

[Breakpoints view](#)

Removing breakpoints

Breakpoints can be easily removed when you no longer need them.

1. In the editor area, open the file where you want to remove the breakpoint.
2. Directly to the left of the line where you want to remove the breakpoint, open the marker bar pop-up menu and select ***Toggle Breakpoint***. The breakpoint is removed from the workbench. You can also double-click directly on the breakpoint icon to remove it.

Breakpoints can also be removed in the ***Breakpoints view***. Select the breakpoint(s) to be removed and from the context menu select ***Remove***.

All breakpoints can be removed from the workbench using the ***Remove All*** action in the context menu of the ***Breakpoints view***.

If you find yourself frequently adding and removing a breakpoint in the same place, consider disabling the breakpoint when you don't need it (using ***Disable Breakpoint*** in the breakpoint context menu or the ***Breakpoints view***) and enabling it when needed again.

■ Related concepts

[Debugger](#)

[Java perspectives](#)

[Java editor](#)

■ Related tasks

[Adding breakpoints](#)

[Enabling and disabling breakpoints](#)

[Applying hit counts](#)

[Catching Java exceptions](#)

[Managing conditional breakpoints](#)

[Setting method breakpoints](#)

[Stepping through the execution of a program](#)

■ Related reference

[Breakpoints view](#)

Enabling and disabling breakpoints

Breakpoints can be enabled and disabled as needed. When a breakpoint is enabled, thread execution suspends before that line of code is executed. When a breakpoint is disabled, thread execution is not suspended by the presence of the breakpoint.

To disable a breakpoint in the *Breakpoints view*:

1. Open the breakpoint's context menu and select **Disable**, or deselect the breakpoint's checkbox.
2. The breakpoint image will change to a white circle and its checkbox will be empty.

To disable a breakpoint in the marker bar of an editor:

1. Open the breakpoint's context menu and select **Disable Breakpoint**.
2. The breakpoint image will change to a white circle.

To enable the breakpoint in the *Breakpoints view*:

1. Open the breakpoint's context menu and select **Enable**, or select the breakpoint's checkbox.
2. The breakpoint image will change back to a blue circle, and its checkbox will be checked.

To enable a breakpoint in the marker bar of an editor:

1. Open the breakpoint's context menu and select **Enable Breakpoint**.
2. The breakpoint image will change to a white circle.

■ Related concepts

[Debugger](#)

[Java perspectives](#)

[Java editor](#)

■ Related tasks

[Applying hit counts](#)

[Catching Java exceptions](#)

[Removing breakpoints](#)

[Setting method breakpoints](#)

[Managing conditional breakpoints](#)

[Stepping through the execution of a program](#)

■ Related reference

[Breakpoints view](#)

Applying hit counts

A hit count can be applied to line breakpoints, exception breakpoints, watchpoints and method breakpoints. When a hit count is applied to a breakpoint, the breakpoint suspends execution of a thread the *n*th time it is hit, but never again, until it is re-enabled or the hit count is changed or disabled.

To set a hit count on a breakpoint:

1. Select the breakpoint to which a hit count is to be added.
2. From the breakpoint's pop-up menu, select ***Hit Count***.
3. In the ***Enter the new hit count for the breakpoint*** field, type the number of times you want to hit the breakpoint before suspending execution.

Note: When the breakpoint is hit for the *n*th time, the thread that hit the breakpoint suspends. The breakpoint is disabled until either it is re-enabled or its hit count is changed.

■ Related concepts

[Breakpoints](#)

■ Related tasks

[Adding breakpoints](#)

[Removing breakpoints](#)

[Enabling and disabling breakpoints](#)

[Setting method breakpoints](#)

■ Related reference

[Breakpoints view](#)

Setting method breakpoints

Method breakpoints are used when working with types that have no source code (binary types).

1. Open the class in the Outline view, and select the method where you want to add a method breakpoint.
2. From the method's pop-up menu, select ***Toggle Method Breakpoint***.
3. A breakpoint appears in the Breakpoints view. If source exists for the class, then a breakpoint also appears in the marker bar in the file's editor for the method that was selected.
4. While the breakpoint is enabled, thread execution suspends when the method is entered, before any line in the method is executed.

Method breakpoints can also be setup to break on method exit. In the Breakpoints view, select the breakpoint and toggle the ***Exit*** item in its context menu.

Method breakpoints can be removed, enabled, and disabled just like line breakpoints.

■ Related concepts

[Breakpoints](#)

■ Related tasks

[Adding breakpoints](#)

[Removing breakpoints](#)

[Enabling and disabling breakpoints](#)

[Applying hit counts](#)

[Catching Java exceptions](#)

■ Related reference

[Breakpoints view](#)

[Java outline](#)

Breakpoints view

The Breakpoints view lists all the breakpoints you have set in the workbench projects. You can double-click a breakpoint to display its location in the editor. In this view, you can also enable or disable breakpoints, delete them, or add new ones.

This view also lists Java exception breakpoints, which suspend execution at the point where the exception is thrown. You can add or remove exceptions.

■ Related concepts

[Breakpoints](#)

[Java views](#)

■ Related tasks

[Adding breakpoints](#)

[Applying hit counts](#)

[Catching Java exceptions](#)

[Removing breakpoints](#)

[Enabling and disabling breakpoints](#)

[Managing conditional breakpoints](#)

[Setting method breakpoints](#)

■ Related reference

[Views and editors](#)

Managing conditional breakpoints

An enabling condition can be applied to line breakpoints, so that the breakpoint suspends execution of a thread in one of these cases:

- when the enabling condition is true
- when the enabling condition changes

To set a condition on a breakpoint:

1. Find the breakpoint to which an enabling condition is to be applied (in the Breakpoints view or in the editor marker bar).
2. From the breakpoint's pop-up menu, select **Breakpoint Properties....** The Breakpoint properties dialog will open.
3. In the properties dialog, check the **Enable Condition** checkbox.
4. In the **Condition** field enter the expression for the breakpoint condition.
5. Do one of the following:
 - ◆ If you want the breakpoint to stop every time the condition evaluates to *true*, select the **condition is 'true'** option. The expression provided must be a boolean expression.
 - ◆ If you want the breakpoint to stop only when the result of the condition changes, select the **value of condition changes** option.
6. Click **OK** to close the dialog and commit the changes. While the breakpoint is enabled, thread execution suspends before that line of code is executed if the breakpoint condition evaluates to *true*.

A conditional breakpoint has a question mark overlay on the breakpoint icon.

■ Related concepts

[Debugger](#)

[Java perspectives](#)

[Java editor](#)

■ Related tasks

[Adding breakpoints](#)

[Applying hit counts](#)

[Catching Java exceptions](#)

[Removing breakpoints](#)

[Setting method breakpoints](#)

[Stepping through the execution of a program](#)

■ Related reference

[Breakpoints view](#)

Views and editors

■ Related concepts

[Java editor](#)

[Java views](#)

[Java Development Tools \(JDT\)](#)

■ Related tasks

[Changing the appearance of the console view](#)

[Changing the appearance of the Hierarchy view](#)

[Opening an editor for a selected element](#)

[Opening an editor on a type](#)

[Using content/code assist](#)

■ Related reference

[Java editor actions](#)

[Breakpoints view](#)

[Console view](#)

[Debug view](#)

[Display view](#)

[Expressions view](#)

[Java outline](#)

[Package Explorer view](#)

[Type Hierarchy view](#)

[Variables view](#)

Changing the appearance of the Hierarchy view

The Hierarchy view offers three different ways to look at a type hierarchy (use the toolbar buttons to alternate between them):

- **Show the Type Hierarchy** displays the type hierarchy of a type. This view shows all super- and subtypes of the selected type. Type *java.lang.Object* is shown in the top-left corner. Interfaces are not shown.
- **Show the Supertype Hierarchy** displays the supertype hierarchy of a type. This view shows all supertypes and the hierarchy of all implemented interfaces. The selected type is always shown in the top-left corner.
- **Show the Subtype Hierarchy** displays the subtype hierarchy of a type. This view shows all subtypes of the selected type and all implementors of the selected interface (if the view is opened on an interface). The selected type is always shown in the top-left corner.

■ Related concepts

[Java views](#)

■ Related tasks

[Using the Hierarchy view](#)

■ Related reference

[Views and editors](#)

[Type Hierarchy view](#)

Using the Hierarchy view

■ Related tasks

[Changing the appearance of the Hierarchy view](#)

[Opening a type hierarchy on a Java element](#)

[Opening a type hierarchy on the current text selection](#)

[Opening a type hierarchy in its own perspective](#)

[Finding overridden methods](#)

■ Related reference

[Views and editors](#)

[Type Hierarchy view](#)

Opening a type hierarchy on a Java element

There are several ways to open a type hierarchy. Select a Java element in a Java view and:

- Press **F4** or
- Choose **Open Type Hierarchy** from the view's pop-up menu or
- Drag and drop the element to the Hierarchy view or
- Press **Ctrl+Shift+H** and select a type from the list in the resulting dialog (you cannot select a package, source folder, or project with this method) or
- Select **Navigate > Open Type in Hierarchy** from the menu bar or
- Select **Focus On...** from the pop-up menu of the type hierarchy viewer.

■ Related concepts

Java development tools (JDT)

■ Related tasks

Changing new type hierarchy defaults

Creating Java elements

Using the Hierarchy view

Opening a type hierarchy on a Java element

Opening a type hierarchy on the current text selection

Opening a type hierarchy in the workbench

Opening a type hierarchy in its own perspective

■ Related reference

Views and editors

Type Hierarchy view

Changing new type hierarchy defaults

New type hierarchies can open in a Hierarchy view or in a Java Hierarchy perspective. You can indicate which is the default method for opening new type hierarchies on the preferences pages.

1. Select **Window > Preferences**, and select the **Java** category. The general Java Preferences page opens.
2. Use the radio buttons that appear under **When opening a Type Hierarchy** to indicate your preference.

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Opening a type hierarchy on a Java element](#)

[Opening a type hierarchy on the current text selection](#)

[Opening a type hierarchy in the workbench](#)

[Opening a type hierarchy in its own perspective](#)

■ Related reference

[Java Base preference page](#)

Opening a type hierarchy on the current text selection

To open a type hierarchy on a text selection, select the name of a Java element in the editor, and do one of the following:

- Press **F4**
- select **Open Type Hierarchy** from the editor's pop-up menu
- select **Navigate > Open Type Hierarchy** from the menu bar

Note: If the selected Java element (or editor selection) is not a type or a compilation unit, the hierarchy opens on the type enclosing the current selection.

■ Related concepts

Java editor

■ Related tasks

Using the Hierarchy view

Changing new type hierarchy defaults

Changing the appearance of the Hierarchy view

Opening a type hierarchy on a Java element

Opening a type hierarchy in the workbench

Opening a type hierarchy in its own perspective

■ Related reference

Views and editors

Type Hierarchy view

Edit menu

Opening a type hierarchy in the workbench

You can open a type hierarchy from a button in the workbench toolbar.

1. In the workbench toolbar, click the *Open Type* button.
2. Select a type in the dialog.
3. Select the *Open in Type Hierarchy* checkbox.
4. Click *OK*.

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Changing new type hierarchy defaults](#)

[Opening a type hierarchy on a Java element](#)

[Opening a type hierarchy on the current text selection](#)

[Opening a type hierarchy in its own perspective](#)

■ Related reference

[Views and editors](#)

[Type Hierarchy view](#)

Opening a type hierarchy in its own perspective

The default behavior for opening the Hierarchy view is to replace the Hierarchy view that is already open in the perspective. If there are no open Hierarchy views, one is opened automatically.

Sometimes, it is useful to look at or work with several type hierarchies in different perspectives. This can be achieved by changing a workbench preference.

1. From the menu bar, select **Window > Preferences** and select the **Java** category. The general Java Preferences page opens.
2. In the **When Opening a Type Hierarchy** category, select the radio button **Open a new Type Hierarchy Perspective**.

■ Related concepts

[Java perspectives](#)

■ Related tasks

[Using the Hierarchy view](#)

[Changing new type hierarchy defaults](#)

[Opening a type hierarchy on a Java element](#)

[Opening a type hierarchy on the current text selection](#)

[Opening a type hierarchy in the workbench](#)

■ Related reference

[Views and editors](#)

[Type Hierarchy view](#)

[Java Base preference page](#)

Type Hierarchy view

This view shows the hierarchy of a type. The Type Hierarchy view consists of two panes:

- Type Hierarchy tree pane
- Member list pane (optional)

Type Hierarchy tree pane toolbar buttons

Command	Description
Previous Hierarchy Inputs	This menu displays a history of previously displayed type hierarchies.
Show the Type Hierarchy	This command displays the type in its full context (i.e., superclasses and subclasses) in the Hierarchy view. To see for which type the hierarchy is shown, hover over the view title (e.g., "Types").
Show the Supertype Hierarchy	<p>This command displays the supertypes and the hierarchy of all implemented interfaces of the type in the Hierarchy view. The tree starts at the selected type and displays the result of traversing up the hierarchy.</p> <p><i>Note: The selected type is always at the top level, in the upper-left corner.</i></p>
Show the Subtype Hierarchy	<p>This command displays the subtypes of the selected class and/or all implementors of the interface in the Hierarchy view. The tree starts at the selected type and displays the result of traversing down the hierarchy</p> <p><i>Note: The selected type is always at the top level, in the upper-left corner.</i></p>
Vertical View Orientation	Arranges the two panes vertically.
Horizontal View Orientation	Arranges the two panes horizontally.
Hierarchy View Only	Hides the member list pane.

Member list pane toolbar buttons

The member list pane displays the members of the currently selected type in the type hierarchy tree pane.

Command	Description
Lock View and Show Members in Hierarchy	<p>Shows the members implementing the selected method Only types implementing the method are shown.</p> <p>When the view is locked, the member list pane no longer tracks the selection in the hierarchy pane above.</p>
Show All Inherited Members	

Basic tutorial

	Shows or hides all methods and fields inherited by base classes. When this option is set, the name of the type that defines the method is appended to the method name.
Sort Members by the Defining Type	Sorts the members according to the type in which they are defined.
Hide Fields	Shows or hides the fields.
Hide Static Members	Shows or hides the static fields and methods.
Hide Non-Public Members	Shows or hides the static fields and methods.

Related reference

Views and editors

Java

On this page, indicate your preferences for the general Java settings.

Java Preferences

Option	Description	Default
Action on double click in the Package Explorer	<p><i>Go into the selected element:</i> When you double click a container, a <i>Go Into</i> command is executed. See <i>Go Into</i> from the Navigate menu.</p> <p><i>Expand the selected element:</i> When you double click a container, it is expanded and its children are revealed.</p>	Expand the selected element
When opening a Type Hierarchy	<p><i>Open a new Type Hierarchy Perspective</i> Opens a new Type Hierarchy perspective whenever a Type Hierarchy view is opened.</p> <p><i>Show the Type Hierarchy View in the current perspective</i> The Type Hierarchy view is displayed in the current perspective.</p> <p><i>Note: On the Workbench preferences page, you can choose whether new perspectives open in a new window, in the current window, or as a replacement for the current perspective.</i></p>	Show the Type Hierarchy View in the current perspective
Refactoring Java Code	<p><i>Save all modified resources automatically prior to refactoring</i> If this option is turned on, all refactorings prompt whether any modified files should be saved before opening the refactoring wizard.</p>	On
Search	<p><i>Use reduced search menu</i> If this option is turned on, the search context menus show only the most frequently used search actions.</p>	On

■ Related concepts

[Java views](#)

■ Related tasks

[Using the Package Explorer](#)

[Using the Hierarchy view](#)

■ Related reference

Package Explorer view
Hierarchy view

Navigate actions

Navigate menu commands:

Name	Function	Keyboard Shortcut
Go Into	Sets the view input to the currently selected element. Supported by the <u>Packages Explorer view</u> .	
Go To	<ul style="list-style-type: none"> • Back: Sets the view input to the input back in history: Only enabled when a history exists (<i>Go Into</i> was used) • Forward: Sets the view input to the input forward in history: Only enabled when a history exists (<i>Go Into</i>, <i>Go To > Back</i> were used) • Up One Level: Sets the input of the current view to its input's parent element • Referring Tests: Browse for all JUnit tests that refer to the currently selected type • Type: Browse for a type and reveal it in the current view. Supported by the Package Explorer view • Package: Browse for a package and reveal it in the current view. Supported by the Package Explorer view • Resource: Browse for a resource and reveal it in the current view. 	
Open	Tries to resolve the element referenced at the current code selection and opens the file declaring the reference.	F3
Open Type Hierarchy	Tries to resolve the element referenced at the current code selection and opens the element in the <u>Type Hierarchy view</u> . Invoked on elements, opens the type hierarchy of the element. Supported in the Java editor and views showing Java elements.	F4
Open Super Implementation	Open an editor for the super implementation of the currently selected method or method surrounding the current cursor position. No editor is opened if no method is selected or the method has no super implementation.	
Open External Javadoc	Opens the Javadoc documentation of the currently selected element or text selection. The location of the Javadoc of a JAR or a project is specified in the <u>Javadoc Location property page</u> on projects or JARs. Note that this external Javadoc documentation may not be up to date with the Javadoc specified in the current code. You can create Javadoc documentation for source files in a Java project using the <u>Javadoc export wizard</u> .	Shift + F2
Open Type	Brings up the Open Type dialog to open a type in the editor. The Open Type selection dialog shows all types existing in the workspace.	Ctrl + Shift + T
Open Type In Hierarchy	Brings up the Open Type dialog to open a type in the editor and the <u>Type Hierarchy view</u> . The Open Type selection dialog shows all types that exist in the workspace.	Ctrl + Shift + H

Basic tutorial

Show in > Package Explorer	Reveals the currently selected element (or the element surrounding the current cursor position) in the Package Explorer view .	
Show Outline	Opens the lightweight outliner for the currently selected type.	Ctrl + O
Go to Next Problem	Selects the next problem. Supported in the Java editor.	Ctrl + .
Go to Previous Problem	Selects the previous problem. Supported in the Java editor.	Ctrl + ,
Go to Last Edit Location	Reveal the location where the last edit occurred.	Ctrl + Q
Go to Line	Opens an a dialog which allows entering the line number to which the editor should jump to. Editor only.	Ctrl + L

■ Related concepts

[Java views](#)

[Java development tools \(JDT\)](#)

■ Related tasks

[Opening an editor for a selected element](#)

[Showing an element in the Package Explorer](#)

[Opening a type in the Package Explorer](#)

[Opening an editor on a type](#)

[Opening a package](#)

[Opening a type hierarchy on a Java element](#)

[Opening a type hierarchy on the current text selection](#)

[Opening a type hierarchy in the workbench](#)

[Opening a type hierarchy in its own perspective](#)

■ Related reference

[Package Explorer view](#)

[Type Hierarchy view](#)

[Javadoc Location properties](#)

[Javadoc export wizard](#)

Package Explorer view

The Package Explorer view, shown by default in the Java perspective, shows the Java element hierarchy of the Java projects in your workbench. It provides you with a Java-specific view of the resources shown in the Navigator. The element hierarchy is derived from the project's build paths.

For each project, its source folders and referenced libraries are shown in the tree. You can open and browse the contents of both internal and external JAR files. Opening a Java element inside a JAR opens the CLASS file editor, and if there is a source attachment for the JAR file, its corresponding source is shown.

Toolbar buttons

Command	Description
Back	Navigates to the most recently–displayed state of the view with a different element at the top level.
Forward	Navigates to the state of the view with a different element at the top level that was displayed immediately after the current state.
Up	Navigates to the parent container of the package that is currently displayed at the top level in the view.
Collapse All	Collapses all tree nodes.
Link with Editor	Links the package explorer's selection to the active editor.
Select Working Set...	Opens the <i>Select Working Set</i> dialog to allow selecting a working set.
Deselect Working Set	Deselects the current working set.
Edit Active Working Set	Opens the <i>Edit Working Set</i> wizard..
Hide Non–Public Members	Shows or hides the static fields and methods.
Hide Static Members	Shows or hides the static fields and methods.
Hide Fields	Shows or hides fields.
Filters...	Opens the <i>Java Element Filters</i> dialog. <u>See Java Element Filters dialog</u>

■ Related concepts

[Java views](#)

[Java perspectives](#)

■ Related tasks

[Using the Package Explorer](#)

[Showing and hiding elements](#)

■ Related reference

[Java Element Filters dialog](#)

[Views and editors](#)

Java element filters dialog

This dialog lets you define Java element filters for the Package Explorer view.

Option	Description	Default
Name filter patterns	If enabled, a comma separated list of patterns can be specified additionally.	Off
Select the elements to exclude from the view	List of pre-defined filters which can be enabled.	. * files Empty parent packages Import declarations Inner class files Package declarations

The *Filter description* field displays the description for the currently selected filter.

Related tasks

[Filtering elements](#)

[Showing and hiding elements](#)

Related reference

[Package Explorer view](#)

Filtering elements

To filter elements:

1. On the Package Explorer toolbar, click the **Menu** button and choose **Filters**.
2. Select or clear the filters that you want to apply to the view (read **Filter description** to learn about the selected filter's functionality).
3. Optionally, you can select patterns for filtering the view:
 - ◆ Select the **Name filter patterns** checkbox at the top of the dialog.
 - ◆ In the text field below, specify the desired patterns (names matching one of the patterns will be hidden).

■ Related concepts

[Java projects](#)

■ Related tasks

[Using the Package Explorer](#)
[Showing and hiding system files](#)

■ Related reference

[Java Element Filters](#)
[Package Explorer](#)

Using the Package Explorer view

■ Related tasks

[Filtering elements](#)

[Showing and hiding elements](#)

[Moving folders, packages and files](#)

■ Related reference

[Package Explorer](#)

Showing and hiding elements

You can use filters to control which files are displayed in the Package Explorer.

■ Related tasks

[Showing and hiding system files](#)

[Showing and hiding CLASS files generated for inner types](#)

[Showing and hiding libraries](#)

[Showing single elements or whole Java files](#)

[Showing and hiding empty packages](#)

[Showing and hiding empty parent packages](#)

[Showing and hiding Java files](#)

[Showing and hiding non-Java elements](#)

[Showing and hiding non-Java projects in Java views](#)

[Showing and hiding members in Java views](#)

[Showing and hiding override indicators](#)

[Showing and hiding method return types in Java views](#)

[Showing and hiding import declarations in Java views](#)

[Showing and hiding package declarations in Java views](#)

■ Related reference

[Java Element Filters](#)

[Package Explorer](#)

Showing and hiding system files

To show system files:

1. Select the ***Filters*** command from the Package Explorer view drop–down menu.
2. In the exclude list clear the checkbox for `.*` files.

To hide system files:

1. Select the ***Filters*** command from the Package Explorer view drop–down menu.
2. In the exclude list select the checkbox for `.*` files. This hides files that have only a file extension but no file name, such as `.classpath`.

■ Related tasks

[Showing and hiding elements](#)

[Filtering elements](#)

■ Related reference

[Java Element Filters](#)

[Package Explorer](#)

Showing and hiding CLASS files generated for inner types

To show CLASS files for inner types:

1. Select the ***Filters*** command from the Package Explorer drop-down menu.
2. Ensure that the ***Inner class files*** filter is not selected.

To hide CLASS files for inner types:

1. Select the ***Filters*** command from the Package Explorer drop-down menu.
2. Ensure that the ***Inner class files*** filter is selected.

■ Related tasks

[Showing and hiding elements](#)

[Filtering elements](#)

■ Related reference

[Java Element Filters](#)

[Package Explorer](#)

Showing and hiding libraries

To show libraries:

1. Select the ***Filters*** command from the Package Explorer's drop-down menu.
2. In the exclude list clear the checkbox for Referenced libraries.

To hide libraries:

1. Select the ***Filters*** command from the Package Explorer's drop-down menu.
2. In the exclude list select the checkbox for Referenced libraries.

■ Related tasks

[Showing and hiding elements](#)

[Filtering elements](#)

■ Related reference

[Java Element Filters](#)

[Package Explorer](#)

Showing single element or whole Java file

- To display the selected Java file in a single element view, click the *Show Source of Selected Element Only* button in the workbench toolbar, so that it is pressed.
- To display the selected Java file in a whole (non-segmented) view, click the *Show Source of Selected Element Only* button in the workbench toolbar, so that it is not pressed.

Note: this toolbar button is enabled only when a Java editor is open.

■ Related concepts

[Java editor](#)

■ Related tasks

[Using the Java editor](#)

■ Related reference

[Java editor](#)

Java editor

Toolbar actions

Command	Description
Show Source of Selected Element Only	<p>This option can be toggled to display a segmented view of the source of the selected Java element. This button applies to the currently-active editor and to all editors opened in the future; other currently-open editors are not affected.</p> <p>For example, if a method is selected in the Outline view, the Show Source Of Selected Element Only option causes only that method to be displayed in the editor, as opposed to the entire class.</p> <p>Off:</p> <p>The entire compilation unit is displayed in the editor, with the selected Java element highlighted in the marker bar with a range indicator.</p> <p>On:</p> <p>Only the selected Java element is displayed in the editor, which is linked to the selection in the Outline or Hierarchy view.</p>
Go to Next Problem	This command navigates to the next problem marker in the active editor.
Go to Previous Problem	This command navigates to the previous problem marker in the active editor.

Key binding actions

The following actions can only be reached through key bindings. The **Key bindings** field in **Window > Preferences > General > Keys** must be set to 'Emacs'.

Key binding	Description
Alt+0 Ctrl+K, Esc 0 Ctrl+K	Deletes from the cursor position to the beginning of the line.
Ctrl+K	Deletes from the cursor position to the end of the line.
Ctrl+Space, Ctrl+2	Sets a mark at the current cursor position.
Ctrl+X Ctrl+X	Swaps the cursor and mark position if any.

■ Related concepts

[Java editor](#)

■ Related tasks

[Using the Java editor](#)

[Viewing documentation and information](#)

[Showing single elements or whole Java files](#)

[Opening an editor for a selected element](#)

■ Related reference

[Java outline](#)

[Java editor preferences](#)

[JDT actions](#)

[Views and editors](#)

Viewing documentation and information

You can view different kinds of documentation information while working in the workbench.

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Using the Java editor](#)

[Viewing Javadoc information](#)

[Viewing marker help](#)

Viewing Javadoc information

The JDT provides easy access to Javadoc information for the code edited in the Java editor.

1. Open the Java editor on a Java file.
2. Place the mouse pointer over the element whose Javadoc information you want to view (a method name, for example).
3. If Javadoc information is available, a pop-up window opens, displaying the Javadoc information. HTML Javadoc can be viewed as any other resource in the workbench, through an embedded or external editor or viewer. Import the Javadoc into the workbench and double-click it in the Package Explorer.

You can also view Javadoc information by:

1. Opening the Java editor on a Java file.
2. Placing the mouse pointer over the element whose Javadoc information you want to view (a method name, for example).
3. Pressing **F2** or selecting *Edit > Show Tooltip Description* from the menu bar.

To view Javadoc in the Javadoc view:

1. Open the Java editor on a Java file.
2. Place the caret over the element whose Javadoc information you want to view (a method name, for example).
3. Open the Javadoc View (press **Alt+Shift+Q**, **J** or select *Window > Show View > Javadoc*).

To view Javadoc in an external browser:

1. Open the Java editor on a Java file.
2. Place the caret over the element whose Javadoc information you want to view (a method name, for example).
3. Press **Shift+F2** or select *Navigate > Open External Javadoc* from the menu bar.

■ Related concepts

[Java editor](#)

■ Related tasks

[Using the Java editor](#)

[Using content/code assist](#)

[Viewing documentation and information](#)

[Viewing marker help](#)

■ Related reference

[Package Explorer](#)

[Javadoc Location](#)

Using content/code assist

You can use content assist (also called code assist) when writing Java code or Javadoc comments.

1. Place your cursor in a valid position on a line of code in an editor and either
 - ◆ Press **Ctrl+Space**
 - ◆ Select **Edit > Content Assist** from the menu barIf the Java editor finds valid candidates for this position, a list of possible completions is shown in a floating window. You can type further to narrow the list. You can also hover over the selected list items to view its Javadoc information, if it is available.
2. Use the arrow keys or the mouse pointer to navigate through the possible completions.
3. Select an item in the list to complete the fragment by doing one of the following:
 - ◆ Selecting it and pressing **Enter**
 - ◆ Double-clicking it
 - ◆ *Note: When a list item is selected, you can view any available Javadoc information for this item in hover help. Note that you must click an item to select it in the list before you can view Javadoc hover help for it.*

■ Related concepts

[Java editor](#)

[Scrapbook](#)

■ Related tasks

[Using the Java editor](#)

[Formatting Java code](#)

[Using the Java editor](#)

[Viewing Javadoc information](#)

[Views and editors](#)

■ Related reference

[Java Content Assist](#)

Scrapbook

The JDT contributes a scrapbook facility that can be used to experiment and evaluate Java code snippets before building a complete Java program. Snippets are edited and evaluated in the Scrapbook page editor, with resultant problems reported in the editor.

From a Java scrapbook editor, you can select a code snippet, evaluate it, and display the result as a string. You can also show the object that results from evaluating a code snippet in the debugger's expressions view.

■ Related concepts

[Java development tools \(JDT\)](#)

[Debugger](#)

■ Related tasks

[Creating a Java scrapbook page](#)

[Displaying the result of evaluating an expression](#)

[Inspecting the result of evaluating an expression](#)

[Using content/code assist](#)

[Viewing compilation errors and warnings](#)

■ Related reference

[New Java Scrapbook Page wizard](#)

[Java scrapbook page](#)

[Expressions view](#)

Creating a Java scrapbook page

The scrapbook allows Java expressions, to be run, inspected, and displayed under the control of the debugger. Breakpoints and exceptions behave as they do in a regular debug session.

Code is edited on a scrapbook page. A VM is launched for each scrapbook page in which expressions are being evaluated. The first time an expression is evaluated in a scrapbook page after it is opened, a VM is launched. The VM for a page will remain active until the page is closed, terminated explicitly (in the debugger or via the *Stop the Evaluation* button in the editor toolbar), or when a *System.exit()* is evaluated.

There are several ways to open the New Java Scrapbook Page wizard.

- Create a file with a *.jpage* extension
- From the menu bar, select **File > New > Other**. Then select **Java > Java Run/Debug > Scrapbook Page**. Then click *Next*.

Once you've opened the New Java Scrapbook Page wizard:

1. In the *Enter or select the folder field*, type or click **Browse** to select the container for the new page.
2. In the *File name* field, type a name for the new page. The *.jpage* extension will be added automatically if you do not type it yourself.
3. Click **Finish** when you are done. The new scrapbook page opens in an editor.

■ Related concepts

[Scrapbook](#)

[Java projects](#)

■ Related tasks

[Creating a new source folder](#)

[Creating Java elements](#)

[Running and debugging](#)

■ Related reference






[Java scrapbook page](#)

Java scrapbook page

The scrapbook allows Java expressions to be run, inspected, and displayed, under the control of the debugger.

Note: Content assist (such as code assist) is available on scrapbook pages.

Java Scrapbook page buttons

Command	Name	Description
	Run Snippet	Running an expression evaluates an expression but does not display a result.
	Display	Displaying shows the result of evaluating an expression as a string in the scrapbook editor.
	Inspect	Inspecting shows the result of evaluating an expression in the Expressions view.
	Terminate	This command terminates the Java VM that is used to evaluate expressions.
	Set the Import Declarations	This commands sets the import declarations to be used for the context of evaluating the code

■ Related concepts

Scrapbook

■ Related tasks

Creating a Java scrapbook page

Displaying the result of evaluating an expression

Inspecting the result of evaluating an expression

Executing an expression

■ Related reference

New Java Scrapbook Page wizard

Displaying the result of evaluating an expression

Displaying shows the result of evaluating an expression in the scrapbook editor.

1. In the scrapbook page, either type an expression or highlight an existing expression to be displayed.
For example: `System.getProperties()` ;
2. Click the **Display** button in the toolbar (or select **Display** from the selection's pop-up menu.)
3. The result of the evaluation appears highlighted in the scrapbook editor. The result displayed is either
 - ◆ the value obtained by sending `toString()` to the result of the evaluation, or
 - ◆ when evaluating a primitive data type (e.g., an `int`), the result is the simple value of the result.

For example:

- Type and highlight new `java.util.Date()` in the editor, and click **Display**. A result such as `(java.util.Date) Tue Jun 12 14:03:17 CDT 2001` appears in the editor.
- As another example, type and highlight `3 + 4` in the editor, and press **Display**. The result `(int) 7` is displayed in the editor.

■ Related concepts

[Scrapbook](#)

■ Related tasks

[Executing an expression](#)

[Inspecting the result of evaluating an expression](#)

[Viewing runtime exceptions](#)

■ Related reference

[Java scrapbook page](#)

Executing an expression

Executing an expression evaluates an expression but does not display a result.

If you select the expression to execute and click the ***Execute*** button in the toolbar, no result is displayed, but the code is executed.

For example, if you type and highlight `System.out.println("Hello World")`, and click the ***Execute*** button, *Hello World* appears in the Console view, but no result is displayed in the scrapbook editor or the Expressions view.

■ Related concepts

[Java views](#)

■ Related tasks

[Displaying the result of evaluating an expression](#)

[Inspecting the result of evaluating an expression](#)

[Viewing runtime exceptions](#)

■ Related reference

[Expressions view](#)

[Console view](#)

Inspecting the result of evaluating an expression

Inspecting shows the result of evaluating an expression in the Expressions view.

1. In the scrapbook page, either type an expression or highlight an existing expression to be inspected.
For example: `System.getProperties();`
2. Click the **Inspect** button in the toolbar (or select **Inspect** from the selection's pop-up menu).
3. The result of the inspection appears in a pop-up.
4. The result can be inspected like a variable in the debugger (for example, children of the result can be expanded).

■ Related concepts

[Scrapbook](#)

■ Related tasks

[Creating a Java scrapbook page](#)

[Displaying the result of evaluating an expression](#)

[Executing an expression](#)

[Viewing runtime exceptions](#)

■ Related reference

[Expressions view](#)

[Java scrapbook page](#)

Viewing runtime exceptions

If an expression you evaluate causes a runtime exception, the exception will be reported in the editor. For example:

Type and select the expression `Object x = null; x.toString()` in the editor and click **Display** in the toolbar.

The error message:

An exception occurred during evaluation: java.lang.NullPointerException

will be displayed in the editor.

■ Related concepts

[Java editor](#)

■ Related tasks

[Displaying the result of evaluating an expression](#)

[Inspecting the result of evaluating an expression](#)

[Executing an expression](#)

Expressions view

Data can be inspected in the Expressions view. You can inspect data from a scrapbook page, a stack frame of a suspended thread, and other places. The Expressions view opens automatically when an item is added to the view.

■ Related concepts

[Java views](#)

[Java perspectives](#)

[Scrapbook](#)

■ Related tasks

[Evaluating expressions](#)

[Suspending threads](#)

■ Related reference

[Views and editors](#)

New Java Scrapbook Page Wizard

This wizard helps you to create a new Java scrapbook page in a project.

Create Java Scrapbook Page Options

Option	Description	Default
Enter or select the parent folder	Type or browse the hierarchy below to select a container (project or folder) for the scrapbook page.	The container of the selected element
File name	Type a name for the new file. The ".jpage" extension is appended automatically when not added already.	<blank>

■ [Related reference](#)

[Java Scrapbook page File actions](#)

Viewing compilation errors and warnings

The workbench supports different ways to examine errors and warnings:

- Error ticks in Java views (e.g. Package Explorer)
- Marker annotations in the editor
- Problems in the Problems view
- Tasks in the Tasks view

If an expression you select to evaluate has a compilation error, it will be reported in the Scrapbook editor.

For example, if you type and select the (invalid) expression `System.println("hi")` in the editor and click **Run** in the toolbar, the error message:

The method println(java.lang.String) is undefined for the type java.lang.System

is displayed in the editor at the point of the error.

■ Related concepts

[Java builder](#)

[Java editor](#)

[Scrapbook](#)

■ Related tasks

[Building automatically](#)

[Building manually](#)

[Running and debugging](#)

[Setting execution arguments](#)

■ Related reference

[Package Explorer](#)

Setting execution arguments

If you want to specify execution arguments for your program, you must define a launch configuration that specifies the arguments.

1. Select **Run >Run...** (or **Run >Debug...**) from the workbench **Run** menu to open the list of launch configurations. Launch configurations for Java programs are shown underneath **Java Application** in this list.
2. Create a new launch configuration by pushing the **New** button after selecting **Java Application**.
3. On the **Arguments** tab for the configuration, you can specify the following fields as necessary:
 - ◆ **Program Arguments:** Application-specific values that your code is expecting (a user name or a URL for locating help files, for example).
 - ◆ **VM Arguments:** Values meant to change the behavior of the Java virtual machine (VM). For example, you may need to tell the VM whether to use a just-in-time (JIT) compiler, or you may need to specify the maximum heap size the VM should use. Refer to your VM's documentation for more information about the available VM arguments.
 - ◆ **Working Directory:** The working directory used for the launched process. To change from using the default working directory, uncheck **Use default working directory** and specify the workspace or local directory to use for the working directory of the launched process.
4. Click **Apply** or **Close** when you are done. Every time you launch this configuration, these execution arguments will be used.

■ Related tasks

[Creating a Java Application launch configuration](#)

[Launching a Java program](#)

Creating a Java application launch configuration

When you choose **Run >Run As >Java Application** to launch your class, you are running your class using a generic **Java Application** launch configuration that derives most of the launch parameters from your Java project and your workbench preferences. In some cases, you will want to override the derived parameters or specify additional arguments.

You do this by creating your own **Java Application** launch configuration.

1. Select **Run >Run...** or **Run >Debug...** from the workbench menu bar. This opens a dialog that lets you create, modify, and delete launch configurations of different types.
2. Select **Java Application** in the left hand list of launch configuration types, and press **New**. This will create a new launch configuration for a Java application. The tabs on the right hand side allow you control specific aspects of the launch.

- ◆ The **Main** tab defines the class to be launched. Enter the name of the project containing the class to launch in the project field, and the fully qualified name of the main class in the the Main class field. Check the **Stop in main** checkbox if you want the program to stop in the main method whenever the program is launched in debug mode.
Note: You do not have to specify a project, but doing so allows a default classpath, source lookup path, and JRE to be chosen.
- ◆ The **Arguments** tab defines the arguments to be passed to the application and to the virtual machine (if any). You can also specify the working directory to be used by the launched application.
- ◆ The **JRE** tab defines the JRE used to run or debug the application. You can select a JRE from the already defined JREs, or define a new JRE.
- ◆ The **Classpath** tab defines the location of class files used when running or debugging an application. By default, the user and bootstrap class locations are derived from the associated project's build path. You may override these settings here.
- ◆ The **Source** tab defines the location of source files used to display source when debugging a Java application. By default, these settings are derived from the associated project's build path. You may override these settings here.
- ◆ The **Environment** tab defines the environment variable values to use when running or debugging a Java application. By default, the environment is inherited from the Eclipse runtime. You may override or append to the inherited environment.
- ◆ The **Common** tab defines general information about the launch configuration. You may choose to store the launch configuration in a specific file and specify which perspectives become active when the launch configuration is launched.

■ Related concepts

[Debugger](#)

[Local debugging](#)

■ Related tasks

[Choosing a JRE for launching a project](#)

[Launching a Java program](#)

[Setting execution arguments](#)

[Changing debugger launch options](#)

■ Related reference

[Debug preferences](#)

[Debug view](#)

[Run and debug actions](#)

Changing the active perspective when launching

You can control which perspective becomes active when a program is launched and when it suspends. The setting is configurable for each launch configuration type, for each of the launch modes it supports.

To activate a particular perspective when a program is launched, do the following:

1. Open the debugger launch options preferences page (**Window > Preferences > Run/Debug > Launching**).
2. Select the **Always** option for the **Open the associated perspective when launching** preference. This will cause the perspective associated with a program to become active whenever it is launched.

To activate a particular perspective when a program is suspends, do the following:

1. Open the debugger preferences page (**Window > Preferences > Run/Debug**).
2. Select the **Always** option for the **Open the associated perspective when a breakpoint is hit** preference. This will cause the perspective associated with a program to become active whenever a program suspends.

To associate a particular perspective with a program, do the following:

1. Open the run dialog (**Run > Run...**).
2. Select the type of launch configuration (program) that you would like to associate a perspective with (for example, **Java Application**). The **Perspectives** tab will be displayed.
3. For each launch mode, select the desired perspective using the combo box. This will cause the perspective you choose to become active based on your preference settings (i.e. when a program is launched and/or when it suspends).
4. Press the **Apply** button to save the settings.

If the specified perspective is not open at the time it needs to be activated, that perspective is created.

■ Related concepts

[Debugger](#)
[Remote debugging](#)
[Local debugging](#)
[Java perspectives](#)

■ Related tasks

[Running and debugging](#)
[Setting execution arguments](#)
[Launching a Java program](#)

■ Related reference

[Console view](#)
[Debug preferences](#)
[Debug view](#)

Run and debug actions

Debug preferences

The following preferences can be set using the Debug Preferences page.

Option	Description	Default
Build (if required) before launching	If the workspace requires building, an incremental build will be performed prior to launching an application.	On
Remove terminated launches when a new launch is created	When an application is launched, all terminated applications in the Debug view are automatically cleared.	On
Reuse editor when displaying source code	The debugger displays source code in an editor when stepping through an application. When this option is on, the debugger will reuse the editor that it opened to display source from different source files. This prevents the debugger from opening an excessive number of editors. When this option is off, the debugger will open a new editor for each source file that needs to be displayed.	On
Activate the workbench when a breakpoint is hit	This option brings attention to the debugger when a breakpoint is encountered, by activating the associated window. The visual result varies from platform to platform. For example, on Windows, the associated window's title bar will flash.	On
Save dirty editors before launching	<p>This option controls whether the user will be prompted to save any dirty editors before an application is launched. The allowable settings are:</p> <ul style="list-style-type: none">• Never – when this option is selected, the user is never prompted to save dirty editors, and editors are not automatically saved.• Prompt – when this option is selected, the user is prompted to save dirty editors before launching an application.• Auto-save – when this option is selected, any dirty editors are automatically saved before launching (and the user is not prompted).	Prompt

■ Related concepts

[Debugger](#)

[Local Debugging](#)

[Remote Debugging](#)

■ Related tasks

[Changing Debugger Launch Options](#)

[Preparing to Debug](#)

[Running and Debugging](#)

■ Related reference

[Java search tab](#)

[Search menu](#)

Preparing to debug

You can make your programs easier to debug by following these guidelines:

- Where possible, do not put multiple statements on a single line, because some debugger features operate on a line basis. For example, you cannot step over or set line breakpoints on more than one statement on the same line.
- Attach source code to JAR files if you have the source code.

■ Related concepts

[Debugger](#)

[Remote debugging](#)

[Local debugging](#)

■ Related tasks

[Changing debugger launch options](#)

[Running and debugging](#)

■ Related reference


[Debug preferences](#)

[Debug view](#)

[Run and debug actions](#)

Run and debug actions

Run and Debug Actions

Toolbar Button	Command	Description
	Run	This command re-launches the most recently launched application.
	Debug	This command re-launches the most recently launched application under debugger control.
Run Menu	Debug Last Launched	This command allows you to quickly repeat the most recent launch in debug mode (if that mode is supported).
Run Menu	Run Last Launched	This command allows you to quickly repeat the most recent launch in run mode (if that mode is supported).
Run Menu	Run History	Presents a sub menu of the recent history of launch configurations launched in run mode
Run Menu	Run As	Presents a sub menu of registered run launch shortcuts. Launch shortcuts provide support for workbench or active editor selection sensitive launching.
Run Menu	Run...	This command realizes the launch configuration dialog to manage run mode launch configurations.
Run Menu	Debug History	Presents a sub menu of the recent history of launch configurations launched in debug mode.
Run Menu	Debug As	Presents a sub menu of registered debug launch shortcuts. Launch shortcuts provide support for workbench or active editor selection sensitive launching.
Run Menu	Debug...	This command realizes the launch configuration dialog to manage debug mode launch configurations.
Run Menu	Various step commands	These commands allow you to step through code being debugged.
Run Menu	Inspect	When a thread suspends, this command uses the Expressions view to show the result of inspecting the selected expression or variable in the context of a stack frame or variable in that thread.
Run Menu	Display	When a thread suspends, this command uses the Display view to show the result of evaluating the selected expression in the context of a stack frame or variable in that thread. If the current active part is a Java Snippet Editor, the result is displayed there.
Run Menu	Run Snippet	Within the context of the Java snippet editor, this command allows you to evaluate an expression but does not display a result.
Run Menu	Run to Line	When a thread is suspended, it is possible to resume execution until a specified line is executed. This is a convenient way to suspend execution at a line without setting a breakpoint.

Basic tutorial

Run Menu	Toggle Line Breakpoint	This command allows you to add or remove a Java line breakpoint at the current selected line in the active Java editor.
Run Menu	Add Java Exception Breakpoint	This command allows you to create an exception breakpoint. It is possible to suspend the execution of thread or VM when an exception is thrown by specifying an exception breakpoint. Execution can be suspended at locations where the exception is uncaught, caught, or both.
Run Menu	Toggle Method Breakpoint	This command allows you to add or remove a method breakpoint for the current binary method. The binary method can be selected in source of a Java class file editor, or be selected in any other view (such as the Outline view).
Run Menu	Toggle Watchpoint	This command allows you to add or remove a field watchpoint for the current Java field. The field can be selected in the source of a Java editor, or be selected in any other view (such as the Outline view).

■ Related concepts

[Debugger](#)

[Local Debugging](#)

[Remote Debugging](#)

■ Related tasks

[Running and Debugging](#)

[Connecting to a remote VM with the Remote Java application launch configuration](#)

[Line breakpoints](#)

[Setting method breakpoints](#)

[Catching exceptions](#)

■ Related reference

[Debug View](#)

[Debug Preferences](#)

[Run and Debug actions](#)

Java search tab

This tab in the Search dialog allows you to search for Java elements.

Search string

In this field, type the expression for which you wish to search, using the wildcard characters mentioned in the dialog as needed. This field is initialized based on the current selection.

- Depending on what is searched for, the search string should describe the element:
 - ◆ Type: the type name (may be qualified or not).
Examples:
 - ◇ *org.eclipse.jdt.internal.core.JavaElement*
 - ◇ *MyClass.Inner*
 - ◇ *Foo*
 - ◆ Method: the defining type name (may be qualified or not as for Type search, optional), the method selector and its parameters (optional).
Examples:
 - ◇ *org.eclipse.jdt.internal.core.JavaElement.getHandleFromMemento(MementoTokenizer, WorkingCopyOwner)*
 - ◇ *equals(Object)*
 - ◇ *foo*
 - ◆ Package: the package name for a package (e.g. *org.eclipse.jdt.internal.core*)
 - ◆ Constructor: the defining type name (may be qualified or not as for Type search, optional) and the constructor parameters (optional).
Examples:
 - ◇ *org.eclipse.jdt.internal.core.JavaElement(JavaElement, String)*
 - ◇ *Foo*

Note that the constructor name should not be entered as it is always the same as the type name.
 - ◆ Field: the defining type name (qualified or not as for Type search, optional) and the field name.
Examples:
 - ◇ *org.eclipse.jdt.internal.core.JavaElement.name*
 - ◇ *foo*
- From the drop-down menu, you can choose to repeat (or modify) a recent search.
Select the Case sensitive field to force a case aware search. Case sensitive is enabled when a custom search string is entered.

Search For

Select to search for one of the following kinds of elements:

- Type
- Method
- Package
- Constructor
- Field

Limit To

Select to limit your search results to one of the following kinds of matches:

- Declarations
- Implementors (available only when searching for types)
- References
- All Occurrences
- Read Access (available only when searching for fields)
- Write Access (available only when searching for fields)

If you would like to search the JRE system libraries as well, select the *Search the JRE system libraries* checkbox.

Scope

Select to limit your search results to one of the following scope

- Workspace
- Selected Resources
- Working Set

Press Choose to select or create a working set.

Related concepts

Java search

Related tasks

Conducting a Java search using the search dialog

Conducting a Java search using pop-up menus

Related reference

Search

Java search

The Java searching support allows you to find declarations, references and occurrences of Java elements (packages, types, methods, fields). Searching is supported by an index that is kept up to date in the background as the resources corresponding to Java elements are changed. The Java search operates on workspaces independent of their build state. For example, searches can be conducted when auto-build is turned off.

The following searches can be initiated from the pop-up menus of Java elements:

Command	Description
References	Finds all references to the selected Java element
Declarations	Finds all declarations of the selected Java element
Implementors	Finds all implementors of the selected Java interface
Read Access	Finds all read accesses to the selected Java field
Write Access	Finds all write accesses to the selected Java field
Occurrences in File	Finds all occurrences of the selected Java element in its file

The scope of the search is defined as:

Workspace – all projects and files in the workspace are included in this search

Enclosing Projects – the projects enclosing the currently selected elements

Hierarchy – only the type hierarchy of the selected element is included in this search

Working Set – only resources that belong to the chosen working set are included in this search

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Searching Java code](#)

■ Related reference

[Search actions](#)

[Java Search tab](#)

Searching Java code

A Java search can be conducted using the Search dialog as well as using the pop-up menu of selected resources and elements.

■ Related concepts

Java search

■ Related tasks

Conducting a Java search using pop-up menus

Conducting a Java search using the Search button

■ Related reference

Search menu

Conducting a Java search using pop-up menus

Open the context menu on any Java element visible in a view.

- Search, Outline, and Hierarchy views: The selected Java element in these views can be searched for declarations and references.
- Package Explorer: Packages, Java compilation units, types and their members can be searched for declarations and references. If a compilation unit or CLASS file contains more than one type, a dialog prompts you to choose one.

The search pop-up menu is also available in the Java editor. If the selection in the Java editor can be resolved to a Java element, then you can search for declarations and references.

To conduct a search from a pop-up menu, follow these steps:

1. Select a Java element (for example a Java compilation unit in the Package Explorer or a method in the Outline view) or some text in a Java editor.
2. From the selection's pop-up menu, navigate to the available Java searches. After you select a search to perform, the search progress is shown in a dialog. Note: in the editor searches are available under the **Search** submenu.
3. You may stop the search process by clicking **Cancel** in the progress dialog.

The type of the selected Java element defines which search pop-up menus are available. The Java editor does not constrain the list of available Java searches based on the selection.

■ Related concepts

[Java search](#)

■ Related tasks

[Searching Java code](#)

■ Related reference

[Java search tab](#)

[Search menu](#)

Search actions

Search menu commands:

Name	Function	<i>Keyboard Shortcut</i>
Search...	Opens the search dialog	Ctrl + H
File...	Opens the search dialog on the File search page	
Help...	Opens the search dialog on the Help search page	
Java...	Opens the search dialog on the Java search page	
References	Finds all references to the selected Java element	
Declarations	Finds all declarations of the selected Java element	
Implementors	Finds all implementors of the selected interface.	
Read Access	Finds all read accesses to the selected field	
Write Access	Finds all write accesses to the selected field	
Occurrences in File	Finds all occurrences of the selected Java element in its file	Ctrl + Shift + U

Search Scopes Submenu:

Scope	<i>Availability</i>	Description
Workspace	all elements	Searches in the full workspace
Project	all elements	Searches in the project enclosing the selected element
Hierarchy	types and members	Searches in the type's hierarchy
Workings Set	all elements	Searches in a working set

Scopes can be saved and names in the working set dialog. Existing instances of working sets are also available in the Search Scope submenu

A Java search can also be conducted via the context menu of selected resources and elements in the following views:

- Package Explorer
- Outline view
- Search result view
- Hierarchy view
- Browsing views

The search context menu is also available in the Java editor. The search is only performed if the currently selected text can be resolved to a Java element.

The type of the selected Java element defines which search context menus are available. The Java editor does not constrain the list of available Java searches based on the selection.

■ Related concepts

Java search

■ Related tasks

Conducting a Java search using the search dialog

Conducting a Java search using pop-up menus

■ Related reference

Java Search tab

Conducting a Java search using the Search dialog

Java search allows you to quickly find references to and declarations of Java elements.

- Open the Search dialog by either:
 - ◆ Clicking the **Search** button in the toolbar or
 - ◆ Pressing **Ctrl+H** or
 - ◆ Selecting **Search > Search...** from the menu bar.
- Select the **Java Search** tab.
- In the **Search string** field, type the string for which you want to search, using wildcards as needed.
- You can also choose a previous search expression from the drop-down list. Selecting a previous search expression restores all values associated with that previous search in the dialog.
- Select the Java element type in the **Search For** area.
- Narrow your search in the **Limit To** area, or select **All occurrences** to search for references and declarations to a Java element.
- Optionally, select the **Search the JRE system libraries** checkbox to include the JRE in your search.
- Optionally, use the **Scope** area to narrow the scope of your search.
- Click **Search**, and the search is carried out. The results are displayed in the Search view in the workbench window.
- Optionally, use the Search view's view menu to switch between flat and hierarchical result presentation, or to configure filters.

■ Related concepts

Java search

■ Related tasks

Conducting a Java search using pop-up menus

■ Related reference

Java search tab

Search menu

Formatting Java code

The Java editor supports the formatting of Java code according to your personal preferences.

■ Related concepts

[Java development tools \(JDT\)](#)

[Java editor](#)

■ Related tasks

[Using the Java editor](#)

[Setting code formatting preferences](#)

[Formatting files or portions of code](#)

[Using content/code assist](#)

■ Related reference

[Code Formatter preferences](#)

Setting code formatting preferences

1. From the workbench menu bar, select **Window > Preferences**. The Workbench Preferences page opens.
2. In the left pane, expand the **Java** category and select **Code Formatter**. The Code Formatter Preferences page opens.
3. Select a profile and click the **Show** or **Edit** button to configure it, or press **New** to create a new profile.
4. In the dialog, select the code formatting conventions that you want the formatter to follow.
5. Note that at the right side of the page, you can observe an example effect of each individual code formatting option and see a preview of what your formatted code will look like.
6. Click **OK** when you are done.

Note: Code formatter settings can also be configured per project:

1. Select a java project, open the pop-up menu and choose **Properties**.
2. Select the **Code Style > Formatter** page and check **Enable project specific sttings**.
3. Select or edit a profile as explained above.
4. Click **OK** when you are done.

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Formatting Java code](#)

[Formatting files or portions of code](#)

■ Related reference

[Code Formatter preferences](#)

Formatting files or portions of code

To format Java code:

1. Use the Code Formatting Preferences page (*Window > Preferences > Java > Code Style > Formatter*) to specify your preferences for code formatting.
2. Open a Java file and select the code you want to format. If nothing is selected, then all of the editor content is formatted.
3. Format the code by either
 - ◆ Selecting *Source > Format* from the editor's pop-up menu or
 - ◆ Pressing *Ctrl+Shift+F* or
 - ◆ Selecting *Source > Format* from the menu bar.

■ Related concepts

[Java development tools \(JDT\)](#)

[Java editor](#)

■ Related tasks

[Using the Java editor](#)

[Formatting Java code](#)

[Setting code formatting preferences](#)

■ Related reference

[Source menu](#)

[Code Formatter preferences](#)

Source actions

Source menu commands:

Name	Function	<i>Keyboard Shortcut</i>
Toggle Comment	Comments or uncomments all lines containing the current selection.	Ctrl + Shift + C
Add Block Comment	Adds a block comment around all lines containing the current selection.	Ctrl + Shift + /
Remove Block Comment	Removes a block comment from all lines containing the current selection.	Ctrl + Shift + \
Shift Right	Increments the level of indentation of the currently select lines. Only activated when the selection covers multiple lines or a single whole line.	
Shift Left	Decrements the level of indentation of the currently select lines. Only activated when the selection covers multiple lines or a single whole line.	
Format	Uses the code formatter to format the current text selection. The formatting options are configured on the Code Formatter preference page (Window > Preferences > Java > Code Style > Formatter)	Ctrl + Shift + F
Format Element	Uses the code formatter to format the Java element comprising the current text selection. The Format Element action works on method and type level. The formatting options are configured on the Code Formatter preference page (Window > Preferences > Java > Code Style > Formatter)	
Correct Indentation	Corrects the indentation of the line denoted by the current text selection.	Ctrl + I
Sort Members	Sorts the members of a type according to the sorting order specified in (Window > Preferences > Java > Appearance > Members Sort Order)	
Organize Imports	Organizes the import declarations in the compilation unit currently open or selected. Unnecessary import declarations are removed, and required import declarations are ordered as specified in the Organize Imports preference page (Window > Preferences > Java > Organize Imports). Organize imports can be executed on incomplete source and will prompt you when a referenced type name can not be mapped uniquely to a type in the current project. You can also organize multiple compilation units by invoking the action on a package or selecting a set of compilation units.	Ctrl + Shift + O
Add Import	Creates an import declaration for a type reference currently selected. If the type reference is qualified, the qualification will be removed if	Ctrl + Shift + M

Basic tutorial

	possible. If the referenced type name can not be mapped uniquely to a type of the current project you will be prompted to specify the correct type. Add Import tries to follow the import order as specified in the Organize Import preference page	
Override/Implement Methods	Opens the Override Method dialog that allows you to override or implement a method in the current type. Available on types or on a text selection inside a type.	
Generate Getter and Setter	Opens the Generate Getters and Setters dialog that allows you to create Getters and Setters for fields in the current type. Available on fields and types or on a text selection inside a type.	
Generate Delegate Methods	Opens the Generate Delegate Methods dialog that allows you to create method delegates for fields in the current type. Available on fields.	
Generate Constructor using Fields	Adds constructors which initialize fields for the currently selected types. Available on types, fields or on a text selection inside a type.	
Add Constructor from Superclass	Adds constructors as defined in the super class for the currently selected types. Available on types or on a text selection inside a type.	
Add Comment	Adds constructors as defined in the super class for the currently selected types. Available on types or on a text selection inside a type.	Alt + Shift + J
Surround with try/catch	For the selected statements all exception that have to be caught are evaluated. A try catch block is created around these expressions. You can use <i>Expand Selection to</i> from the Edit menu to get a valid selection range.	
Externalize Strings	Opens the Externalize strings wizard. This wizards allows you to replace all strings in the code by statements accessing a property file.	
Find Strings to Externalize	Shows a dialog presenting a summary of number of strings not externalized. Available on projects, source folders and packages.	

■ Related concepts

[Java editor](#)

[String externalization](#)

[Java development tools \(JDT\)](#)

■ Related tasks

[Using the Java editor](#)

[Externalizing Strings](#)

■ Related reference

[Java editor](#)

[Java editor preferences](#)

[Java outline](#)

Views and editors

Code Formatter

This page lets you manage your code formatter profiles for the Java code formatter.

Action	Description
New...	Shows the dialog to create a new formatter profile. The dialog requires you to enter a name for the new formatter profile. Additionally, you may select a built-in or user-defined existing formatter profile to base your new formatter profile on.
Show/Edit...	Shows a dialog which displays the settings stored in the selected code formatter profile. Only user-defined profiles can be edited.
Rename...	Renames the selected code formatter profile. This action is only available on user-defined profiles.
Remove	Removes the selected code formatter profile. This action is only available on user-defined profiles.
Import...	Imports code formatter profiles from the file system.
Export...	Exports the selected code formatter profile to the file system. This action is only available on user-defined profiles.

■ Related tasks

[Formatting Java code](#)

■ Related reference

[Java editor](#)

[Java editor preferences](#)

Java editor

The following Java editor preferences can be set on this page and its sub-pages.


- [Appearance and Navigation](#)
- [Code Assist](#)
- [Syntax Coloring](#)

Note that some options that are generally applicable to text editors can be configured on the text editor preference page.

Appearance and Navigation

Java editor appearance and navigation options.

Appearance and Navigation

Option	Description	Default
Smart caret positioning at line start and end	If enabled, the Home and End commands jump to the first and last non whitespace character on a line.	On
Smart caret positioning in Java names (overrides platform behavior)	If enabled, there are additional word boundaries inside <code> Camel Case </code> Java names.	On
Report problems as you type	If enabled, the editor marks errors and warnings as you type, even if you do not save the editor contents. The problems are updated after a short delay.	On
Highlight matching brackets	<p>If enabled, whenever the cursor is next to a parenthesis, bracket or curly braces, its opening or closing counter part is highlighted.</p> <p>The color of the bracket highlight is specified with <i>Appearance color options</i>.</p>	On
Light bulb for quick assists	If enabled, a  shows up in the vertical ruler whenever a quick assist is available. See the quick assist section for a list of the available assists.	Off
Appearance color options	<p>The colors of various Java editor appearance features are specified here.</p> <p>Matching brackets highlight The color of brackets highlight.</p> <p>Find scope The color of find scope.</p> <p>Completion proposal background The background color of the completion proposal window</p>	default colors

Basic tutorial

	<p>Completion proposal foreground The foreground color of the completion proposal window</p> <p>Method parameter background The background color of the parameter window</p> <p>Method parameter foreground The foreground color of the parameter window</p> <p>Completion overwrite background The background color of the completion overwrite window</p> <p>Completion overwrite foreground The foreground color of the completion overwrite window</p>	
--	--	--

Code assist

Configures *Code Assist* behavior.

Code Assist

Option	Description	Default
Completion inserts/Completion overwrites	<p>If Completion inserts is on, the completion text is inserted at the caret position, so it never overwrites any existing text.</p> <p>If Completion overwrites is on, the completion text replaces the characters following the caret position until the end of the word.</p> <p>Note that pressing Ctrl when applying a completion proposal toggles between the two insertion modes.</p>	Completion inserts
Insert single proposals automatically	If enabled, code assist will choose and insert automatically single proposals.	On
Insert common prefixes automatically	If enabled, code assist will automatically insert the common prefix of all possible completions similar to unix shell expansion. This can be used repeatedly, even while the code assist window is being displayed.	Off
Automatically add import instead of qualified name	If enabled, type proposals which are in other packages will invoke the addition of the corresponding import declaration. Otherwise, the type will be inserted fully qualified.	On
Fill argument names on completion	If enabled, code assist will add the argument names when completing a method or generic type.	Off
Guess filled method arguments	If enabled, code assist will try to guess method parameters from the context where a method proposal is inserted.	Off
Present proposals in alphabetical order	If enabled, the proposals are sorted in alphabetical order.	Off
		On

Basic tutorial

Hide proposals not visible in the invocation context	If enabled, the Java element proposals are limited by the rules of visibility. For example, private field proposals of other classes would not be displayed.	
Hide forbidden references	If enabled, references to Java elements forbidden by access rules are not displayed.	On
Hide discouraged references	If enabled, references to Java elements discouraged by access rules are not displayed.	Off
Enable auto activation	If enabled, code assist can be invoked automatically. The condition for automatic invocation is specified with the preferences <i>Auto activation delay</i> , <i>Auto activation triggers for Java</i> and <i>Auto activation triggers for Javadoc</i> .	On
Auto activation delay	If the time starting when an auto activation trigger character is encountered until a new character is typed exceeds the auto activation delay, code assist is invoked.	200
Auto activation triggers for Java	If one of the trigger characters is typed inside Java source code (but not inside a Javadoc comment) and no other character is typed before the auto activation delay times out, the code assist is invoked.	'.'
Auto activation triggers for Javadoc	If one of the trigger characters is typed inside a Java doc and no other character is typed before the auto activation delay times out, the code assist is invoked.	'@#'

Syntax Coloring

Syntax Coloring specifies how Java source code is rendered. Note that general text editor settings such as the background color can be configured on the general 'Text Editors' preference pages. Fonts may be configured on the general 'Colors and Fonts' preference page.

Syntax Coloring

Option	Description	Default
Element	Each category (Java , Javadoc and Comments) contains a list of language elements that may be rendered with its own color and style. Note that some semantic highlighting options can be disabled by the user in order to ensure editor performance on low-end systems.	default colors and styles
Preview	Displays the preview of a Java source code respecting the current colors and styles.	n/a

■ Related concepts

Java Editor

■ Related tasks

Using the Java editor

■ Related reference

Java editor

Code Formatter preferences

Java outline

Java Content Assist

Quick Fix

List of Quick Assists

Quick assists perform local code transformations. They are invoked on a selection or a single cursor in the Java editor and use the same shortcut is used as for quick fixes (Ctrl + 1), but quick assist are usually hidden when an error is around.

A selection of quick assist can be assigned to a direct shortcut. By default these are:

- Rename in file: Ctrl + 2 + R
- Assign to local: Ctrl + 2 + L
- Assign to field: Ctrl + 2 + F

Assign more shortcuts or change the default shortcuts on the keys preference page.

A quick assist light bulb can be turned on on the Java Editor preference page.

Name	Code example			Invocation location
Inverse if statement	<code>if (x) a(); else b();</code>	>	<code>if (!x) b(); else a();</code>	on 'if' statements with 'else' block
Inverse boolean expression	<code>a && !b</code>	>	<code>!a b</code>	on a boolean expression
Remove extra parentheses	<code>if ((a == b) && (c != d)) {}</code>	>	<code>if (a == b && c != d) {}</code>	on selected expressions
Add paranoid parentheses	<code>if (a == b && c != d) {}</code>	>	<code>if ((a == b) && (c != d)) {}</code>	on selected expressions
Join nested if statements	<code>if (a) { if (b) {} }</code>	>	<code>if (a && b) {}</code>	on a nested if statement
Swap nested if statements	<code>if (a) { if (b) {} }</code>	>	<code>if (b) { if (a) {} }</code>	on a nested if statement
Split if statement with and'ed expression	<code>if (a && b) {}</code>	>	<code>if (a) { if (b) {} }</code>	on an and'ed expression in a 'if'
Split if statement with or'd expression	<code>if (a b) x();</code>	>	<code>if (a) x(); if (b) x();</code>	on an or'd expression in a 'if'
Inverse	<code>x ? b : c</code>	>	<code>!x ? c : b</code>	on a

Basic tutorial

conditional expression			conditional expression
Pull negation up	<code>b && c</code>	<code>> !(b !c)</code>	On a boolean expression
Push negation down	<code>!(b && c)</code>	<code>> !b !c</code>	On a negated boolean expression
If-else assignment to conditional expression	<code>if (a) x= 1; else x= 2;</code>	<code>> x= a ? 1 : 2;</code>	on an 'if' statement
If-else return to conditional expression	<code>if (a) return 1; else return 2;</code>	<code>> return a ? 1 : 2;</code>	on an 'if' statement
Conditional expression assignment to If-else	<code>x= a ? 1 : 2;</code>	<code>> if (a) x= 1; else x= 2;</code>	on a conditional expression
Conditional expression return to If-else	<code>return a ? 1 : 2;</code>	<code>> if (a) return 1; else return 2;</code>	on a conditional expression
Switch to If-else	<code>switch (kind) { case 1: return -1; case 2: return -2; }</code>	<code>> if (kind == 1) { return -1; } else if (kind == 2) { return -2; }</code>	on a switch statement
Exchange operands	<code>a + b</code>	<code>> b + a</code>	on an infix operation
Cast and assign	<code>if (obj instanceof Vector) { }</code>	<code>> if (obj instanceof Vector) { Vector vec= (Vector)obj; }</code>	on a instanceof expression in an 'if' or 'while' statement
Pick out string	<code>"abcdefgh"</code>	<code>> "abc" + "de" + "fgh"</code>	select a part of a string literal
Split variable	<code>int i= 0;</code>	<code>> int i; i= 0;</code>	On a variable with initialization

Basic tutorial

Join variable	<code>int i; i= 0;</code>	>	<code>int i= 0</code>	On a variable without initialization
Assign to variable	<code>foo()</code>	>	<code>X x= foo();</code>	On an expression statement
Extract to local	<code>foo(<u>getColor()</u>);</code>	>	<code>Color color= getColor(); foo(color);</code>	On an expression
Assign parameter to field	<code>public A(int color) {}</code>	>	<code>Color fColor; public A(int color) { fColor= color; }</code>	On a parameter
Add finally block	<code>try { } catch (Expression e) { }</code>	>	<code>try { } catch (Expression e) { } finally {}</code>	On a try/catch statement
Add else block	<code>if (a) b();</code>	>	<code>if (a) b(); else { }</code>	On a if statement
Replace statement with block	<code>f (a) b();</code>	>	<code>if (a) { b(); }</code>	On a if statement
Invert equals	<code>a.equals(b)</code>	>	<code>b.equals(a)</code>	On a invocation of 'equals'
Array initializer to Array creation	<code>int[] i= { 1, 2, 3 }</code>	>	<code>int[] i= new int[] { 1, 2, 3 }</code>	On an array initializer
Convert to 'enhanced for loop' (J2SE 5.0)	<code>for (Iterator i= c.iterator(); i.hasNext();) { }</code>	>	<code>for (x : c) { }</code>	On a for loop
Create method in super class				On a method declaration
Unwrap blocks	<code>{ a() }</code>	>	<code>a()</code>	On blocks, if/while/for statements
Rename in file				On identifiers

■ Related concepts

[Java editor](#)

[Quick Fix](#)

■ Related reference

[JDT actions](#)

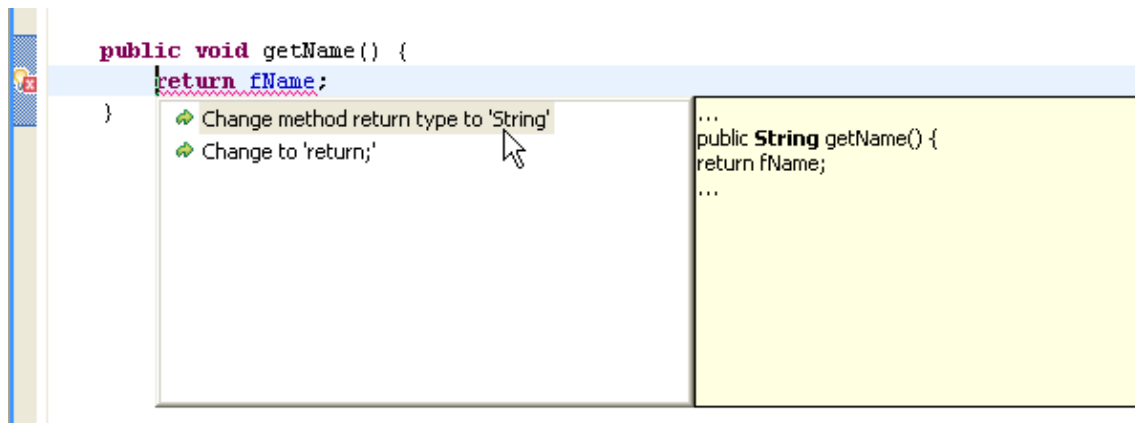
Quick Fix

The Java editor offers corrections to problems found while typing and after compiling. To show that correction proposals are available for a problem or warning, a 'light bulb' is visible on the editor's annotation bar.

Left click on the light bulb or invoking **Ctrl+1 (Edit > Quick Fix)** brings up the proposals for the problem at the cursor position.

Each quick fix show a preview when selected in the proposal window.

Some selected quick fixes can also be assigned with direct shortcuts. You can configure these shortcuts on the keys preference page.



The following quick fixes are available:

Package Declaration

- Add missing package declaration or correct package declaration
- Move compilation unit to package that corresponds to the package declaration

Imports

- Remove unused, unresolvable or non-visible import
- Invoke 'Organize imports' on problems in imports

Types

- Create new class, interface, enum, annotation or type variable for references to types that can not be resolved
- Change visibility for types that are accessed but not visible
- Rename to a similar type for references to types that can not be resolved
- Add import statement for types that can not be resolved but exist in the project
- Add explicit import statement for ambiguous type references (two import-on-demands for the same type)
- If the type name is not matching with the compilation unit name either rename the type or rename the compilation unit
- Remove unused private types

Basic tutorial

Constructors

- Create new constructor for references to constructors that can not be resolved (this, super or new class creation)
- Reorder, add or remove arguments for constructor references that mismatch parameters
- Change method with constructor name to constructor (remove return type)
- Change visibility for constructors that are accessed but not visible
- Remove unused private constructor
- Create constructor when super call of the implicit default constructor is undefined, not visible or throws an exception
- If type contains unimplemented methods, change type modifier to 'abstract' or add the method to implement

Methods

- Create new method for references to methods that can not be resolved
- Rename to a similar method for references to methods that can not be resolved
- Reorder or remove arguments for method references that mismatch parameters
- Correct access (visibility, static) of referenced methods
- Remove unused private methods
- Correct return type for methods that have a missing return type or where the return type does not match the return statement
- Add return statement if missing
- For non-abstract methods with no body change to 'abstract' or add body
- For an abstract method in a non-abstract type remove abstract modifier of the method or make type abstract
- For an abstract/native method with body remove the abstract or native modifier or remove body
- Change method access to 'static' if method is invoked inside a constructor invocation (super, this)
- Change method access to default access to avoid emulated method access

Fields and variables

- Correct access (visibility, static) of referenced fields
- Create new fields, parameters, local variables or constants for references to variables that can not be resolved
- Rename to a variable with similar name for references that can not be resolved
- Remove unused private fields
- Correct non-static access of static fields
- Add 'final' modifier to local variables accessed in outer types
- Change field access to default access to avoid emulated method access
- Change local variable type to fix a type mismatch
- Initialize a variable that has not been initialized

Exception Handling

- Remove unneeded catch block
- Handle uncaught exception by surrounding with try/catch or adding catch block to a surrounding try block
- Handle uncaught exception by adding a throw declaration to the parent method or by generalize an existing throw declaration

Basic tutorial

Build Path Problems

- Add a missing JAR or library for an unresolvable type
- Open the build path dialog for access restriction problems or missing binary classes.
- Change project compliance and JRE to 5.0
- Change workspace compliance and JRE to 5.0

Others

- Add cast or change cast to fix type mismatches
- Let a type implement an interface to fix type mismatches
- For non-NLS strings open the NLS wizard or mark as non-NLS
- Add missing @Override, @Deprecated annotations
- Suppress a warning using @SuppressWarnings

Quick Assists are proposals available even if there is no problem or warning. See the [Quick Assist](#) page for more information.

■ Related concepts

[Java editor](#)

[Quick Assist](#)

■ Related reference

[JDT actions](#)

JDT actions

JDT actions are available from

- Menu bar
- Toolbar
- Context menus in views

■ Related concepts

[Java Development Tools \(JDT\)](#)

■ Related reference

[Frequently Asked Questions on JDT](#)

[JDT Glossary](#)

[File actions](#)

[Edit actions](#)

[Source actions](#)

[Refactor actions](#)

[Navigate actions](#)

[Search actions](#)

[Project actions](#)

[Run actions](#)

[Java Toolbar actions](#)

[Java Editor actions](#)

[Run and Debug actions](#)

Frequently asked questions on JDT

Can I use a Java compiler other than the built-in one (javac for example) with the workbench?

No. The JDT provides a number of sophisticated features including fully automatic incremental recompilation, code snippet evaluation, code assist, type hierarchies, and hot code replace. These features require special support found in the workbench Java compiler (an integral part of the JDT's incremental project builder), but not available in standard Java compilers.

Where do Java packages come from?

A project contains only files and folders. The notion of a Java package is introduced by a Java project's class path (at the UI, the Package Explorer presents the packages as defined by the classpath). **Tip:** If the package structure is not what you expect, check out your class path. The Java search infrastructure only finds declarations for and references from Java elements on the class path.

When do I use an internal vs. an external JAR library file?

An internal resource resides in some project in the workbench and is therefore managed by the workbench; like other resources, these resources can be version managed by the workbench. An external resource is not part of the workbench and can be used only by reference. For example, a JRE is often external and very large, and there is no need to associate it with a VCM system.

When should I use source folders within a Java project?

Each Java project locates its Java source files via one or more source type entries on the project's class path. Use source folders to organize the packages of a large project into useful grouping, or to keep source code separate from other files in the same project. Also, use source folders if you have files (documentation for example) which need not be on the build path.

What are source attachments, How do I define one?

Libraries are stored as JAR files containing binary class files (and perhaps other resources). These binary class files provide signature information for packages, classes, methods, and fields. This information is sufficient to compile or run against, but contains far less information than the original source code. In order to make it easier to browse and debug binary libraries, there is a mechanism for associating a corresponding source JAR (or ZIP) file with a binary JAR file.

Why are all my resources duplicated in the output folder (bin, for example)?

If your Java project is using source folders, then in the course of compiling the source files in the project, the Java compiler copies non-Java resources to the output folder as well so that they will be available on the class path of the running program. To avoid certain resources to be copied to the output location you can set a resource filter in the Java compiler preferences: **Window > Preferences > Java > Compiler > Building**

How do I prevent having my documentation files from being copied to the project's output folder?

Use source folders and put any resources that you do not want to be copied to the output folder into a separate folder that is not included on the class path. You can also set a resource filter in the Java compiler preferences: *Window > Preferences > Java > Compiler > Building* to for example *.doc.

How do I create a default package?

You don't have to. Files in the root folder of a source folder or project are considered to be in the default package. In effect, every source folder has the capability of having a fragment of the default package.

What is refactoring?

Refactoring means behavior-preserving program transformations. The JDT supports a number of transformations described in Martin Fowler's book *Refactoring: Improving the Design of Existing Code*, Addison Wesley 1999.

When do I use code select/code resolve (F3)?

To find out the Java element that corresponds to a source range with the help of the compiler.

Is the Java program information (type hierarchy, declarations, references, for example) produced by the Java builder? Is it still updated when auto-build is off?

The Java program information is independent from the Java builder. It is automatically updated when performing resource changes or Java operations. In particular, all the functionality offered by the Java tooling (for example, type hierarchies, code assisting, search) will continue to perform accurately when auto-build is off; for example, when doing heavy refactoring which require to turn off the builders, you can still use code assist, which will reflect the latest changes (not yet build). Other than the launching (that is, running and debugging) of programs, the only functionality which requires the Java builder is the evaluation of code snippets.

After reopening a workbench, the first build that happens after editing a Java source file seems to take a long time. Why is that?

The Java incremental project builder saves its internal state to a file when the workbench is closed. On the first build after the project is reopened, the Java incremental project builder will restore its internal state. When this file is large, the user experiences an unusually long build delay.

I can't see a type hierarchy for my class. What can I do?

Check that you have your build class path set up properly. Setting up the proper build class path is an important task when doing Java development. Without the correct build path, you will not be able to compile your code. In addition, you will not be able to search or look at the type hierarchies for Java elements.

How do I turn off "auto compile" and do it manually when I want?

Clear the **Window > Preferences > General > Workspace > Build automatically** checkbox. When you want to build, press **Ctrl+B**, or select **Project > Build All** from the menu bar.

Hint: when you turn "auto compile" off and build manually, you may also want to select the **Window > Preferences > General > Workspace > Safe automatically before build** checkbox.

When I select a method or a field in the Outline view, only the source for that element is shown in the editor. What do I do to see the source of the whole file?

There is a toolbar button **Show Source of Selected Element Only** – all you have to do is un-press it.

Can I nest source folders?

Yes, you can use exclusion filters to create nested source folders.

Can I have separate output folders for each source folder?

Yes, select the **Allow output folders for source folders** checkbox in the **Java Build Path > Source** property page of your Java project.

Can I have an output or source folder that is located outside of the workspace?

Yes, you can create a linked folder that points to the desired location and use that folder as the source or output folder in your Java project.

Related concepts

[Java development tools \(JDT\)](#)

Related reference

[Java Build Path page](#)

[JDT glossary](#)

JDT glossary

CLASS file

A compiled Java source file.

compilation unit

A Java source file.

field

A field inside a type.

import container

Represents a collection of import declarations. These can be seen in the Outline view.

import declaration

A single package import declaration.

initializer

A static or instance initializer inside a type.

JAR file

JAR (Java archive) files are containers for compiled Java source files. They can be associated with an archive (such as, ZIP, JAR) as a source attachment. The children of JAR files are packages. JAR files can be either compressed or uncompressed.

JAVA elements

Java elements are Java projects, packages, compilation units, class files, types, methods and fields.

JAVA file

An editable file that is compiled into a byte code (CLASS) file.

Java projects

Projects which contain compilable Java source code and are the containers for source folders or packages.

JDT

Java development tools. Workbench components that allow you to write, edit, execute, and debug Java code.

JRE

Java runtime environment (for example, J9, JDK, and so on).

method

A method or constructor inside a type.

package declaration

The declaration of a package inside a compilation unit.

packages

A group of types that contain Java compilation units and CLASS files.

refactoring

A comprehensive code editing feature that helps you improve, stabilize, and maintain your Java code. It allows you to make a system-wide coding change without affecting the semantic behavior of the system.

type

A type inside a compilation unit or CLASS file.

source folder

A folder that contains Java packages.

VCM

Version control management. This term refers to the various repository and versioning features in the workbench.

VM

Virtual machine.

■ **Related concepts**

[Java development tools \(JDT\)](#)

■ **Related reference**

[Frequently asked questions on JDT](#)

Edit actions

Edit menu commands shown when a Java Editor is visible:

Name	Function	<i>Keyboard Shortcut</i>
Undo	Revert the last change in the editor	Ctrl + Z
Redo	Revert an undone change	Ctrl + Y
Cut	Copies the currently selected text or element to the clipboard and removes the element. On elements, the remove is not performed before the clipboard is pasted.	Ctrl + X
Copy	Copies the currently selected text or elements to the clipboard	Ctrl + C
Paste	Paste the current content as text to the editor, or as a sibling or child element to the a currently selected element.	Ctrl + V
Delete	Delete the current text or element selection.	Delete
Select All	Select all the editor content..	Ctrl + A
Find / Replace	Open the Find / Replace dialog. Editor only.	Ctrl + F
Find Next	Finds the next occurrence of the currently selected text. Editor only.	Ctrl + K
Find Previous	Finds the previous occurrence of the currently selected text. Editor only.	Ctrl + Shift + K
Incremental Find Next	Starts the incremental find mode. After invocation, enter the search text as instructed in the status bar. Editor only.	Ctrl + J
Incremental Find Previous	Starts the incremental find mode. After invocation, enter the search text as instructed in the status bar. Editor only.	Ctrl + Shift + J
Add Bookmark	Add a bookmark to the current text selection or selected element.	
Add Task	Add a user defined task to the current text selection or selected element.	Alt + Enter
Expand Selection to	<ul style="list-style-type: none"> • Enclosing Element: Selects the enclosing expression, block, method in the code. This action is aware of the Java syntax. It may not function properly when the code has syntax errors. (Arrow Up) • Next Element: Selects the current and next element. (Arrow Right) • Previous Element: Selects the current and the previous element (Arrow Left) • Restore Last Selection: After an invocation of <i>Expand Selection to</i> restore the previous selection. (Arrow Down) 	Alt + Shift + Arrow Keys
Show Tooltip Description	Shows the value of a hover that would appear at the current cursor location. The dialog shown is scrollable and does not shorten descriptions.	F2
Content Assist		

Basic tutorial

	Opens a context assist dialog at the current cursor position to bring up Java code assist proposals and templates. See the Templates preference page for available templates (Window > Preferences > Java > Editor > Templates) and go to the Java Editor preference page (Window > Preferences > Java > Editor > Code Assist) for configuring the behavior of code assist.	Ctrl + Space
Quick Fix	If the cursor is located at a location with problem indication this opens a context assist dialog at the current cursor to present possible corrections.	Ctrl + 1
Parameter Hints	If the cursor is located at the parameter specification for method reference, this actions shows a hover with parameter types information. The parameter at the current cursor location is shown in bold.	Ctrl + Shift + Space
Encoding	Toggles the encoding of the currently shown text content.	

■ Related concepts

[Java editor](#)

[Java development tools \(JDT\)](#)

■ Related tasks

[Using the Java editor](#)

[Using Quick Fix](#)

■ Related reference

[Java editor](#)

[Java editor preferences](#)

[Java outline](#)

[Views and editors](#)

Using Quick Fix

To use the Quick Fix feature:

- You need to have the *Window > Preferences > Java > Editor > Report problems as you type* checkbox selected.
- In the Java editor, if you see an error or a warning underlined with a squiggly line, position the caret inside the underlined range and do one of the following
 - ◆ Press **Ctrl+1** or
 - ◆ From the menu bar, select *Edit > Quick Fix*
- A list of suggested corrections is presented, with a preview displayed when an entry is selected.
- Select an entry from the list and press **Enter**.

Light bulb icons appear on the left-hand side vertical ruler to indicate Quick Fix'able problems. You can then click on one of the the light bulb icons to invoke Quick Fix.

Note: Occasionally, invoking Quick Fix will not suggest any corrections. A message saying '*No suggestions available*' will be displayed in such cases.

Using Quick Assist

The Java editor also offers some common code changes even when there's no error or warning in your code. For example, set the caret into the `if` of the following if-else statement, press **Ctrl+1**, and select "Change 'if-else' statements to blocks":

```
String test(int arg) {  
    if (arg > 1000)  
        return "big";  
    else  
        return "small";  
}
```

Braces are added around the return statements:

```
String test(int arg) {  
    if (arg > 1000) {  
        return "big";  
    } else {  
        return "small";  
    }  
}
```

Check out the [Available Quick Fix proposals](#) and the [Available Quick Assist proposals](#), or just press **Ctrl+1** to see what quick fixes or quick assists Eclipse offers in your current context.

■ Related concepts

[Java editor](#)

[Available Quick Fix proposals](#)

[Available Quick Assist proposals](#)

■ Related tasks

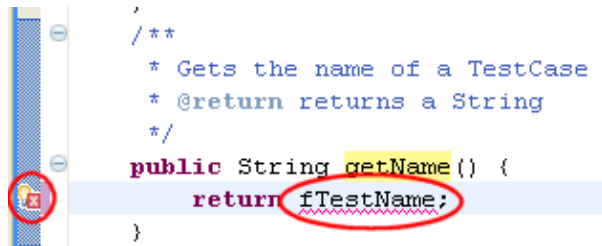
[Using the Java editor](#)

■ Related reference

[Quick Fix](#)

Quick fix

For certain problems underlined with a problem highlight line, the Java editor can offer corrections. This is shown by the light bulb shown in the editor marker bar.



To see the correction proposals use the Quick Fix action

- Set the cursor inside the highlight range, and select **Quick Fix** from the Edit menu or the context menu.
- Set the cursor inside the highlight range, and press Ctrl + 1
- Click on the light bulb

Quick fix is also available on a problem entry in the Problem view. The Quick Fix action will open a dialog to select the correction.

Note that the light bulb is only a hint. It is possible that even with the light bulb shown, it turns out that no corrections can be offered.

A overview of quick fixes available in the Java editor can be found [here](#).

Quick Assists are proposals available even if there is no problem or warning. See the [Quick Assist](#) page for more information.

■ Related concepts

[Quick Fix](#)

[Quick Assist](#)

[Java editor](#)

■ Related tasks

[Using quick fix](#)

■ Related reference

[Java editor preferences](#)

[Edit menu](#)

Java outline

This view displays an outline of the structure of the currently–active Java file in the editor area.

Toolbar buttons

Command	Description
Go into Top Level Type	Makes the top level type of the compilation unit the new input for this view. Package declaration and import statements are no longer shown.
Sort	This option can be toggled to either sort the outline elements in alphabetical order or sequential order, as defined inside the compilation unit. <i>Note: Static members are always listed first.</i>
Hide Fields	Shows or hides the fields.
Hide Static Fields and Methods	Shows or hides the static fields and methods.
Hide Non–Public Members	Shows or hides the static fields and methods.
Hide Local Types	Shows or hides the local types.

■ Related concepts

[Java editor](#)

[Java views](#)

■ Related tasks

[Generating getters and setters](#)

[Restoring a deleted workbench element](#)

[Setting method breakpoints](#)

[Showing and hiding members in Java views](#)

[Showing and hiding override indicators](#)

[Showing and hiding method return types in Java views](#)

[Sorting elements in Java views](#)

■ Related reference

[Override methods](#)

[Views and editors](#)

Restoring a deleted workbench element

1. Ensure that a Java view that show Java elements inside files (such as the Outline view) is visible.
2. Open the compilation unit to which you want to add a previously removed Java element from the local history.
3. Activate the editor by clicking its tab in the editor area, and the Java view shows the content of the Java file.
4. In the Java view, select the element to whose container type you want to restore the deleted element.
5. From the type's pop-up menu in the Java view, select **Restore from Local History**.
6. In the upper left pane of the resulting dialog, all available editions of the selected element in the local history are displayed.
7. In the left pane check all elements that you want to replace.
8. For every checked element select an edition in the right hand pane and view its content in the bottom pane.
9. When you have identified the edition that you want to restore, press **Restore**. The local history editions are loaded into the editor.

■ Related concepts

Java editor

■ Related tasks

Using the local history

■ Related reference

Java outline

Using the local history

The JDT extends the workbench concept of a local history in three ways:

- A file can be replaced with an edition from the local history not just in the Navigator but also in the Package Explorer view.
- The JDT allows you to compare and/or replace individual Java elements (types and their members) with editions from the local history.
- The JDT allows you to restore Java elements (and files) deleted from the workbench that have been kept in the local history.

Note: Files and Java elements such as types and their members change in time. A 'snapshot' of what they look like a point in time (as saved in the local history) is called an edition.

■ Related concepts

[Java development tools \(JDT\)](#)

[Java views](#)

■ Related tasks

[Using the Java editor](#)

[Replacing a Java element with a local history edition](#)

[Comparing a Java element with a local history edition](#)

[Restoring a deleted workbench element](#)

■ Related reference

[Package Explorer](#)

Replacing a Java element with a local history edition

1. Make sure that a Java view is visible.
2. Open a Java editor for the Java file in which you want to replace a Java element with an edition from the local history.
3. Activate the editor by clicking its tab in the editor area. The Outline view also displays the Java file.
*Note: The Package Explorer can be configured to show or not show Java elements in files. Use **Window > Preferences > Java > Appearance > Show Members in Package Explorer** to set your preference.*
4. Select the element that you want to replace in the Outline or the Package Explorer.
5. From the element's pop-up menu, select **Replace With > Element from Local History**.
6. In the upper pane of the resulting dialog, all available editions of the selected element in the local history are displayed.
7. Select an edition in the upper pane to view the differences between the selected edition and the edition in the workbench.
8. When you have identified the edition with which you want to replace the existing Java element, click **Replace**.
9. The local history edition replaces the current one in the editor. *Note: The changed compilation unit has not yet been saved at this point.*

■ Related concepts

[Java views](#)

[Java editor](#)

■ Related tasks

[Using the local history](#)

■ Related reference

[Java outline](#)

Comparing a Java element with a local history edition

1. Make sure that a Java view is visible.
2. Open a Java editor for the Java file in which you want to compare a Java element with an edition from the local history.
3. Activate the editor by clicking its tab in the editor area. The Outline view also displays the Java file.
*Note: The Package Explorer can be configured to show or not show Java elements in files. Use **Window > Preferences > Java > Appearance > Show Members in Package Explorer** to set your preference.*
4. Select the element that you want to compare in the Outline or the Package Explorer.
5. From the element's pop-up menu, select **Compare With > Element from Local History**.
6. In the upper pane of the resulting dialog, all available editions of the selected element in the local history are displayed.
7. Select an edition in the upper pane to view the differences between the selected edition and the edition in the workbench.
8. If you are done with the comparison, click OK to close the dialog.

■ Related concepts

[Java views](#)

[Java editor](#)

■ Related tasks

[Using the local history](#)

■ Related reference

[Java outline](#)

Showing and hiding members

Several Java views (e.g. Outline, Package Explorer, Members) offer filtering of members (fields, types and methods). The filters are available as toolbar buttons or as view menu items, depending on the view. There are 3 member filters:

- **Hide Fields:** when activated, this filter causes all fields to be removed from the view.
- **Hide Static Members:** when activated, this filter causes all static members to be removed from the view.
- **Hide Non-Public Members:** when activated, this filter causes all non-public members to be removed from the view.

Additionally, the Package Explorer can display or hide all elements inside compilation units.

To show members in the Package Explorer:

- Select the Show members in Package Explorer checkbox in the Window > Preferences > Java > Appearance page.

To hide members in the Package Explorer:

- Clear the Show members in Package Explorer checkbox in the Window > Preferences > Java > Appearance page

■ Related tasks

[Showing and hiding elements](#)

[Filtering elements](#)

■ Related reference

[Appearance preference page](#)

[Package Explorer](#)

Appearance

On this preference page, the appearance of Java elements in views can be configured. The options are:

Option	Description	Default
Show method return types	If enabled, methods displayed in views show the return type.	Off
Show method type parameters	If enabled, methods displayed in views show the their type parameters.	On
Show members in Package Explorer	If enabled, Java elements below the level of Java files and Class files are displayed as well.	On
Fold empty packages in hierarchical layout	If enabled, empty packages which do not contain resources or other child elements are folded.	On
Compress package name segments	If enabled, package names, except for the final segment, are compressed according to the compression pattern.	Off
Stack views vertically in the Java Browsing perspective	If enabled, views in Java Browsing perspective will be stacked vertically, rather than horizontally.	Off

■ Related concepts

[Java views](#)

■ Related tasks

[Showing and hiding elements](#)

[Showing full or compressed package names](#)

■ Related reference

[Package Explorer view](#)

Showing full or compressed package names

Package Explorer and Packages views can show full or compressed package names.

To show full package names:

- Clear the Compress all package name segments, except the final segment checkbox on the Window > Preference > Java > Appearance page

To show compressed package names:

- Check the Compress all package name segments, except the final segment checkbox on the Window > Preference > Java > Appearance page

Compression patterns control how many characters of each package name segment are displayed. The last segment of a package name is always displayed.

A compression pattern of "." indicates that only the separating periods are shown to represent a segment. A digit (n) in a compression pattern represents the first n characters of a package name segment. Examples are the best way to understand compression patterns. The package org.eclipse.jdt would be displayed as follows using the example compression patterns:

. ..jdt

0 jdt

2~ or~.ec~.jdt

3~ org.ecl~.jdt

■ Related reference

[Java Appearance preference page](#)
[Package Explorer](#)

Showing and hiding override indicators

Outline and Hierarchy views can show special icons (override indicators) to indicate members that override or implement other members from supertypes.

To show the override indicators:

- Select the Show override indicators in outline and hierarchy checkbox in the Window > Preferences > Java > Appearance page

To hide the override indicators:

- Clear the Show override indicators in outline and hierarchy checkbox in the Window > Preferences > Java > Appearance page

Related tasks

[Showing and hiding elements](#)

[Filtering elements](#)

Related reference

[Appearance preference page](#)

[Package Explorer](#)

Showing and hiding method return types

Several Java views (e.g. Outline, Members) present methods and can also show their return types.

To show method return types in Java views:

- Open the Window > Preferences > Java > Appearance page and select the Show method return types checkbox

To hide method return types in Java views:

- Clear the Show method return types checkbox in the Window > Preferences > Java > Appearance page

■ Related tasks

[Showing and hiding elements](#)

[Filtering elements](#)

■ Related reference

[Appearance preference page](#)

[Package Explorer](#)

Sorting elements in Java views

Members and Outline views can present members sorted or in the order of declaration in the compilation unit.

To sort members:

- Toggle on the Sort toolbar button in the Java view
- The sorting order can be configured on the on the Window > Preferences... > Java > Appearance > Members Sort Order page
- After the above sorting is performed, members in each group are sorted alphabetically

To present members in the order of declaration in the compilation unit.









- Toggle off the Sort toolbar button in the Java view

Related reference

[Java Toolbar actions](#)

Java toolbar actions

Java Actions

Toolbar Button	Command	Description
	Create a Java Project	This command helps you create a new Java project. <u>See New Java Project Wizard</u>
	Create a Java Package	This command helps you create a new Java package. <u>See New Java Package Wizard</u>
	Create a Java Class	This command helps you create a new Java class. <u>See New Java Class Wizard</u>
	Create a Java enum	This command helps you create a new Java enum. <u>See New Java Enum Wizard</u>
	Create a Java Interface	This command helps you create a new Java interface. <u>See New Java Interface Wizard</u>
	Create a Java annotation	This command helps you create a new Java annotation. <u>See New Java Annotation Wizard</u>
	Create a Scrapbook Page	This command helps you create a new Java scrapbook page for experimenting with Java expressions. <u>See New Java Scrapbook Page Wizard</u>
	Open Type	This command allows you to browse the workspace for a type to open in the defined default Java editor. You can optionally choose to display the type simultaneously in the Hierarchy view. <i>Select a type to open:</i>

Basic tutorial

		<p>In this field, type the first few characters of the type you want to open in an editor. You may use wildcards as needed ("?" for any character, "*" for any string, and "TZ" for types containing "T" and "Z" as upper-case letters in camel-case notation, e.g. <code>java.util.TimeZone</code>).</p> <p>Matching Types: This list displays matches for the expression you type in the <i>Select a type to open</i> field.</p>
--	--	---

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Creating Java elements](#)

[Opening an editor on a type](#)

■ Related reference

[New Java Project wizard](#)

[New Java Package wizard](#)

[New Java Class wizard](#)

[New Java Enum wizard](#)

[New Java Interface wizard](#)

[New Java Annotation wizard](#)

[New Java Scrapbook Page wizard](#)

[Views and editors](#)

New Java Package Wizard

This wizard helps you create a folder corresponding to a new Java package. The corresponding folder of the default package always exists, and therefore doesn't have to be created.

Java Package Options

Option	Description	Default
Source folder	Type or browse to select a container (project or folder) for the new package.	The source folder of the element that was selected when the wizard has been started.
Name	Type a name for the new package	<blank>

■ Related tasks

Creating a new Java package

■ Related reference

File actions

Creating a new Java package

To create new Java packages in the Package Explorer:

1. Optionally, select the project or source folder where you want the package to reside.
2. Click the **New Java Package** button in the workbench toolbar. The New Java Package wizard opens.
3. Edit the **Source Folder** field to indicate in which container you want the new package to reside. You can either type a path or click **Browse** to find the container. If a folder was selected when you chose to create the new package, that folder appears in the **Source Folder** field as the container for the new package.
4. In the **Name** field, type a name for the new package.
5. Click **Finish** when you are done.

Note: the default (unnamed) package always exists and does not need to be created.

■ Related concepts

[Java projects](#)

■ Related tasks

[Creating Java elements](#)

[Moving folders, packages, and files](#)

[Organizing Java projects](#)

[Opening a package](#)

[Renaming a package](#)

■ Related reference

[New Java Package wizard](#)

[New Source Folder wizard](#)

[Java Toolbar actions](#)

Moving folders, packages, and files

1. In the Package Explorer, select the folder, package or file you want to move.
2. Drag-and-drop the selected resources onto the desired location.

Note: You can use drag-and-drop to move resources between different workbench windows. Select a resource you want to move, and drag-and-drop it onto the desired destination. If you want to copy rather than move, hold the **Ctrl** key down while dragging.

You can also use drag-and-drop to copy resources between the workbench and the desktop (both ways), but you cannot move files from the desktop to the workbench using drag-and-drop.

■ Related concepts

[Java projects](#)

[Refactoring support](#)

■ Related tasks

[Using the Package Explorer](#)

[Creating a new Java package](#)

[Copying and moving Java elements](#)

[Opening a package](#)

[Renaming a package](#)

■ Related reference

[Package Explorer](#)

[Refactoring actions](#)

Refactoring support

The goal of Java program refactoring is to make system-wide code changes without affecting the behavior of the program. The Java tools provide assistance in easily refactoring code.

The refactoring tools support a number of transformations described in Martin Fowler's book *Refactoring: Improving the Design of Existing Code*, Addison Wesley 1999, such as Extract Method, Inline Local Variable, etc.

When performing a refactoring operation, you can optionally preview all of the changes resulting from a refactoring action before you choose to carry them out. When previewing a refactoring operation, you will be notified of potential problems and will be presented with a list of the changes the refactoring action will perform. If you do not preview a refactoring operation, the change will be made in its entirety and any resultant problems will be shown. If a problem is detected that does not allow the refactoring to continue, the operation will be halted and a list of problems will be displayed.

Refactoring commands are available from the context menus of several Java views (e.g. Package Explorer, Outline) and editors. Many "apparently simple" commands, such as *Move* and *Rename*, are actually refactoring operations, since moving and renaming Java elements often require changes in dependent files.

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Refactoring](#)

[Refactoring without preview](#)

[Refactoring with preview](#)

[Previewing refactoring changes](#)

[Undoing a refactoring operation](#)

[Redoing a refactoring operation](#)

■ Related reference

[Refactoring actions](#)

[Refactoring wizard](#)

[Java preferences](#)

[Extract Method Errors](#)

Refactoring

■ Related concepts

[Refactoring support](#)

■ Related tasks

[Refactoring without preview](#)

[Refactoring with preview](#)

[Previewing refactoring changes](#)

[Undoing a refactoring operation](#)

[Redoing a refactoring operation](#)

[Copying and moving Java elements](#)

[Extracting a method](#)

[Extracting a local variable](#)

[Extracting a constant](#)

[Renaming a package](#)

[Renaming a compilation unit](#)

[Renaming a type](#)

[Renaming a method](#)

[Renaming a field](#)

[Renaming a local variable](#)

[Renaming method parameters](#)

[Changing method signature](#)

[Inlining a local variable](#)

[Inlining a method](#)

[Inlining a constant](#)

[Self encapsulating a field](#)

[Replacing a local variable with a query](#)

[Pulling members up to superclass](#)

[Pushing members down to subclasses](#)

[Moving static members between types](#)

[Moving an instance method to a component](#)

[Converting a local variable to a field](#)

[Converting an anonymous inner class to a nested class](#)

[Converting a nested type to a top level type](#)

[Extracting an interface from a type](#)

[Replacing references to a type with references to one of its supertypes](#)

[Replacing a single reference to a type with a reference to one of its supertypes](#)

[Replacing an expression with a method parameter](#)

[Replacing constructor calls with factory method invocations](#)

[Inferring type parameters for generic type references](#)

■ Related reference

[Refactoring actions](#)

Refactoring dialogs

Java preferences

Refactoring without preview

1. Activate a refactoring command. For example, rename a type by selecting it in the Outline view and choosing **Refactor** > **Rename** from its pop-up menu.
2. The Refactoring Parameters page prompts you for information necessary for the action. For example, the Rename Type Refactoring dialog asks you for a new name for the selected type.
3. Provide the necessary data on the parameters page, and click **OK**.
4. If no problems are anticipated, then the refactoring is carried out.
Otherwise, the problems page comes to the front to display the errors.
5. If a fatal problem was anticipated, then only the **Back** and **Cancel** buttons are enabled, and you are prevented from carrying out the refactoring. If the problems are related to data provided on the parameters page, click **Back** and attempt to remedy the problem. Otherwise, click **Cancel** to close the dialog.
6. If other kinds of problems were anticipated, you can click **Continue** to acknowledge the problems.
The refactoring is carried out, and the dialog closes.

■ Related concepts

[Refactoring support](#)

■ Related tasks

[Refactoring with preview](#)

[Undoing a refactoring operation](#)

[Redoing a refactoring operation](#)

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Refactoring with preview

1. Activate a refactoring command. For example, rename a type by selecting it in the Outline view and choosing **Refactor** > **Rename** from its pop-up menu.
2. The Refactoring Parameters page prompts you for information necessary for the action. For example, the Rename Type Refactoring dialog asks you for the new name for the selected type.
3. Provide the necessary data on the parameters page, and then click **Preview**.
4. If problems are anticipated, then the problems page comes to the front to display them.
 - ◆ If a fatal problem was anticipated, then only the **Back** and **Cancel** buttons are enabled, and you are prevented from carrying out the refactoring. If the problems are related to data provided on the parameters page, click **Back** and attempt to remedy the problem. Otherwise, click **Cancel** to close the dialog.
 - ◆ If other kinds of problems were anticipated, you can click **Continue** to acknowledge the problems and move on to the preview page.The Refactoring Preview page opens.
5. Click **OK** to execute the refactoring and close the dialog.

■ Related concepts

[Refactoring support](#)

■ Related tasks

[Refactoring without preview](#)

[Previewing refactoring changes](#)

[Undoing a refactoring operation](#)

[Redoing a refactoring operation](#)

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Previewing refactoring changes

The Preview page shows the proposed effects of a refactoring action. You can use this page as follows.

- Select a node in the tree to examine a particular change.
- To examine a change inside a compilation unit, expand a compilation unit node in the tree and select one of its children.
- When selecting nodes, the compare viewer is adjusted only to show a preview for the selected node.
- Clear the checkbox for a node to exclude it from the refactoring.

Note: Excluding a node can result in compile errors when performing the refactoring without further warning.

■ Related concepts

[Refactoring support](#)

■ Related tasks

[Refactoring with preview](#)

[Undoing a refactoring operation](#)

[Redoing a refactoring operation](#)

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Undoing a refactoring operation

The most recent refactoring can be undone as long as you have not modified any Java elements in the workbench. After performing a refactoring operation, you can build the project, run and debug it, and execute any test cases, and still undo the refactoring action.

To undo the most recent refactoring, select **Edit > Undo** from the menu bar.

Note: If the workbench contains unsaved files that affect undoing the refactoring, an error dialog appears. The dialog contains a list of files that must be reverted to their saved editions before the refactoring can be completed.

■ Related concepts

[Refactoring support](#)

■ Related tasks

[Refactoring without preview](#)

[Refactoring with preview](#)

[Previewing refactoring changes](#)

[Redoing a refactoring operation](#)

■ Related reference

[Refactoring actions](#)

[Refactoring dialog](#)

[Java preferences](#)

Redoing a refactoring operation

To redo a previously undone refactoring operation, select **Edit > Redo** from the menu bar.

■ Related concepts

[Refactoring support](#)

■ Related tasks

[Refactoring without preview](#)

[Refactoring with preview](#)

[Previewing refactoring changes](#)

[Undoing a refactoring operation](#)

■ Related reference

[Package Explorer](#)

[Java outline](#)

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Refactor actions

Refactor menu commands:

Name	Function	<i>Keyboard Shortcut</i>
Rename	Starts the Rename refactoring dialog: Renames the selected element and (if enabled) corrects all references to the elements (also in other files). Is available on methods, method parameters, fields, local variables, types, type parameters, enum constants, compilation units, packages, source folders, projects and on a text selection resolving to one of these element types.	Alt + Shift + R
Move	Starts the Move refactoring dialog: Moves the selected elements and (if enabled) corrects all references to the elements (also in other files). Can be applied to one instance method (which can be moved to a component), one or more static methods, static fields, types, compilation units, packages, source folders and projects and on a text selection resolving to one of these element types.	Alt + Shift + V
Change Method Signature	Starts the Change Method Signature refactoring dialog. Changes parameter names, parameter types, parameter order and updates all references to the corresponding method. Additionally, parameters can be removed or added and method return type as well as its visibility can be changed. This refactoring can be applied to methods or on text selection resolving to a method.	
Convert Anonymous Class to Nested	Start the Convert Anonymous Class to Nested Class refactoring dialog. Helps you convert an anonymous inner class to a member class. This refactoring can be applied to anonymous inner classes.	
Move Member Type to New File	Starts the Move Member Type to New File refactoring dialog. Creates a new Java compilation unit for the selected member type, updating all references as needed. For non-static member types, a field is added to allow access to the former enclosing instance, if necessary. This refactoring can be applied to member types or text resolving to a member type.	
Push Down	Starts the Push Down refactoring dialog. Moves a set of methods and fields from a class to its subclasses. This refactoring can be applied to one or more methods and fields declared in the same type or on a text selection inside a field or method.	
Pull Up	Starts the Pull Up refactoring wizard. Moves a field or method to a superclass of its declaring class or (in the case of methods) declares the method as abstract in the superclass. This refactoring can be applied on one or more methods, fields and member types declared in the same type or on a text selection inside a field, method or member type.	
Extract Interface	Starts the Extract Interface refactoring dialog. Creates a new interface with a set of methods and makes the selected class implement the interface, optionally changing references to the class to the new interface wherever possible. This refactoring can be applied to types.	

Basic tutorial

Generalize Type	Starts the Generalize Type refactoring dialog. Allows the user to choose a supertype of the reference's current type. If the reference can be safely changed to the new type, it is. This refactoring can be applied to type references and declarations of fields, local variables, and parameters with reference types.	
Use Supertype Where Possible	Starts the Use Supertype Where Possible refactoring dialog. Replaces occurrences of a type with one of its supertypes after identifying all places where this replacement is possible. This refactoring is available on types.	
Infer Generic Type Arguments	Starts the Infer Generic Type Arguments refactoring dialog. Replaces raw type occurrences of generic types by parameterized types after identifying all places where this replacement is possible. This refactoring is available on projects, packages and types.	
Inline	Starts the Inline refactoring dialog. Inlines local variables, methods or constants. This refactoring is available on methods, static final fields and text selections that resolve to methods, static final fields or local variables.	Alt + Shift + I
Extract Method	Starts the Extract Method refactoring dialog. Creates a new method containing the statements or expression currently selected and replaces the selection with a reference to the new method. You can use <i>Expand Selection to</i> from the <u>E</u> dit menu to get a valid selection range. This feature is useful for cleaning up lengthy, cluttered, or overly-complicated methods.	Alt + Shift + M
Extract Local Variable	Starts the Extract Variable refactoring dialog. Creates a new variable assigned to the expression currently selected and replaces the selection with a reference to the new variable. This refactoring is available on text selections that resolve to local variables. You can use <i>Expand Selection to</i> from the <u>E</u> dit menu to get a valid selection range.	Alt + Shift + L
Extract Constant	Starts the Extract Constant refactoring dialog. Creates a static final field from the selected expression and substitutes a field reference, and optionally rewrites other places where the same expression occurs. This refactoring is available on static final fields and text selections that resolve to static final fields.	
Introduce Factory	Starts the Introduce Factory refactoring dialog. This will create a new factory method, which will call a selected constructor and return the created object. All references to the constructor will be replaced by calls to the new factory method. This refactoring is available on constructor declarations.	
Introduce Parameter	Starts the Introduce Parameter refactoring dialog. Replaces an expression with a reference to a new method parameter, and updates all callers of the method to pass the expression as the value of that parameter. This refactoring is available on text selections that resolve to expressions.	
Convert Local Variable to Field	Start the Convert Local Variable to Field refactoring dialog. Turn a local variable into a field. If the variable is initialized on creation, then the operation moves the initialization to the new field's declaration or to the class's constructors. This refactoring is available on text selections that resolve to local variables.	

Encapsulate Field	Starts the Self Encapsulate Field refactoring dialog. Replaces all references to a field with getting and setting methods. Is applicable to a selected field or a text selection resolving to a field.	
-------------------	--	--

Refactoring commands are also available from the context menus in many views and the Java editor.

Related concepts

[Refactoring support](#)

Related tasks

[Refactoring](#)

[Using Structured Selection](#)

Related reference

[Refactoring dialogs](#)

[Extract Method Errors](#)

[Java preferences](#)

Using Structured Selection

Structured Selection lets you quickly select Java code in a syntax-aware way.

To use Structured Selection:

- In a Java editor, (optionally) select some text and press **Alt+Shift+Arrow Up** or select *Edit > Expand Selection To > Enclosing Element* from the menu bar.
- The current text selection is expanded to the inner-most syntax element (more precisely, Abstract Syntax Tree node) that encloses the selection.

When a statement or a list of statements is selected, you can press **Alt+Shift+Arrow Right** or select *Edit > Expand Selection To > Next Element*, which will expand the selection with the statement (if any exists) that is immediately *after* the selected statements.

When a statement or a list of statements is selected, you can press **Alt+Shift+Arrow Left** or select *Edit > Expand Selection To > Previous Element*, which will expand the selection with the statement (if any exists) that is immediately *before* the selected statements.

Pressing **Alt+Shift+Arrow Down** or selecting *Edit > Expand Selection To > Restore Last Selection* from the menu bar lets you restore the previous structured selection.

■ Related concepts

[Java editor](#)

■ Related tasks

[Using the Java editor](#)

[Using Surround with Try/Catch](#)

[Extracting a method](#)

[Extracting a local variable](#)

[Inlining a local variable](#)

[Replacing a local variable with a query](#)

■ Related reference

[Edit menu](#)

Using Surround with Try/Catch

To surround a statement or a set of statements with a try/catch block:

- In the Java editor, select the statement or a set of statements that you want to surround with a try/catch block.
- Do one of the following:
 - ◆ From the menu bar, select **Source > Surround with try/catch Block** or
 - ◆ From the editors pop-up menu, select **Source > Surround with try/catch Block**
- 'catch' blocks for all uncaught exceptions (if there are any) are created. If there are no uncaught exceptions, a dialog appears informing you about this fact and asking if you want to create a 'catch' block for java.lang.RuntimeException.

■ Related concepts

[Java editor](#)

■ Related tasks

[Using the Java editor](#)

[Using Structured Selection](#)

■ Related reference

[Source menu](#)

Extracting a method

To extract a method:

1. In an editor, select a set of statements or an expression from a method body.
2. Do one of the following:
 - ◆ From the pop-up menu in the editor, select ***Refactor > Extract Method***.
 - ◆ From the menu bar, select ***Refactor > Extract Method***.

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Renaming a method](#)

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

[Extract method errors](#)

Renaming a method

You can rename a method by modifying its declaration in the compilation unit in which it is declared. However, if you also want to update all references to it, you must either:

1. In a Java view presenting methods (for example the Outline view) select the method to be renamed.
2. From the view's pop-up menu, select **Refactor > Rename** or select **Refactor > Rename** from the global menu bar.

or

1. In a Java editor, select a reference to or the declaration of the method to be renamed.
2. From the editor's pop-up menu, select **Refactor > Rename** or select **Refactor > Rename** from the global menu bar.

Note 1: Renaming a method declared in an interface also renames (and updates all references to) all methods that are implementations of that method.

Note 2: When renaming a non-static method declared as public, package visible, or protected, all methods overriding it are also renamed and all references to them are updated.

Note 3: Renaming a static method or a private method updates references only to that method.

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Extracting a method](#)

[Renaming method parameters](#)

■ Related reference

[Override methods](#)

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Renaming method parameters

You can rename the parameters of a method by renaming the parameter's declaration as well as all references to the parameters inside the method body.

Use the ***Change Method Signature*** command to rename one or more parameters of a method as well as all references to these parameters.

- Select the method in a Java view (the Outline view, for example)
- From the method's pop-up menu, select ***Refactor > Change Method Signature*** or, from the menu bar, select ***Refactor > Change Method Signature***.
Note: these menu entries will no be active if the method has no parameters.
- Select a parameter, press the ***Edit*** button, enter a new name for the parameter and press ***OK***

To rename a single parameter, it is often easier to:

- Select the parameter in the Java editor.
- From the editor's pop-up menu, select ***Refactor > Rename*** to open the Rename refactoring dialog.

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Changing method signature](#)

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Changing method signature

In addition to renaming a method, you can change other parts of the method's signature.

- Select the method in a Java view (e.g. Outline, Package Explorer, Members).
- Do one of the following to open the Change Method Signature refactoring dialog:
 - ◆ From the menu bar, select ***Refactor > Change Method Signature*** or
 - ◆ From the method's pop-up menu, select ***Refactor > Change Method Signature***

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Renaming a method](#)

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Refactoring Dialog

A dialog based user interface guides you through the steps necessary to execute a selected refactoring. Depending on the complexity of the refactoring, either a wizard or a simple dialog is used to gather information that is required for the refactoring.

- Wizard based user interface (used for example, for *Pull Up*)
- Dialog based user interface (used for example, for *Rename*)

■ Related concepts

Refactoring support

■ Related reference

Refactoring actions

Icons

Wizard based refactoring user interface

A wizard based user interface guides you through the steps necessary to execute a refactoring. A refactoring wizard consists of 1 – n parameter pages, a preview page and a problem page.

Parameter pages

These pages gather information that is required for the refactoring. For example, the *Pull Up* refactoring uses two pages to gather the methods and fields to be pulled up and to gather the obsolete methods and fields in subclasses that can be deleted. The user can navigate the parameter pages using the *Next >* and *< Back* button.

After you have provided the required information, you can click **Finish** to carry out the refactoring without previewing the results. If you want to preview the changes press *Next >*.

Preview page

The JDT allows you to preview the results of a refactoring action before you execute it.

The preview page consists of two parts:

- A tree at the top containing all Java elements affected by the refactoring. Each top–level node in the tree represents one compilation unit.
- A compare viewer at the bottom. The left side of the compare viewer shows the original, the right side displays the refactored source.

Problem page

The Refactoring Problem page indicates if there are suspected, potential, or definite problems with the refactoring action you are attempting.

Four types of problems are possible:

Information

A problem described as Information will not affect the refactoring in any way, nor will it negatively affect the code in the workbench. You can most likely ignore this type of problem.

Warnings

Warnings attempt to predict compiler warnings. This type of problem most likely will not negatively affect the code in your workbench.

Errors

A problem described as an Error is very likely to cause compiler errors or change your workbench code semantically. You can choose to continue with the refactoring in spite of these errors, although it is not recommended.

Stop problems

This type of problem prevents the refactoring from taking place. For example, if you select a comment and choose the Extract Method command from it, the workbench will issue a stop problem on the refactoring attempt because you cannot extract a comment.

Basic tutorial

If there aren't any stop problems then the refactoring can be carried out by pressing the ***Finish*** button. To preview the results of the refactoring action, press the ***Next >*** button.

■ Related concepts

[Refactoring support](#)

■ Related tasks

[Previewing refactoring changes](#)

[Undoing a refactoring operation](#)

[Redoing a refactoring operation](#)
























■ Related reference




















[Refactoring actions](#)

[Icons](#)




JDT icons













Objects

	Compilation Unit (*.java file)
	Java file which is not on a build path
	class file
	file without icon assigned to its type
	unknown object
	Java scrapbook page (*.jpage file)
	Java scrapbook page (evaluation in progress)
	JAR description file
	Java Working Set
	Java Model
	JRE system library container
	JAR file with attached source
	JAR file without attached source
	source folder
	package
	empty package
	logical package
	empty logical package
	package only containing non Java resources
	package declaration
	import container
	import
	default type (package visible)











	public type
	default interface (package visible)
	public interface
	default inner type (package visible)
	private inner type
	protected inner type
	public inner type
	default inner interface (package visible)
	private inner interface
	protected inner interface
	public inner interface
	default field (package visible)
	private field
	protected field
	public field
	default method (package visible)
	private method
	protected method
	public method





Object adornments

	marks project as Java project
	this Java element causes an error
	this Java element causes warning





	this Java element is deprecated
	constructor
	abstract member
	final member
	static member
	synchronized member
	type with public static void main(String[] args)
	implements method from interface
	overrides method from super class
	type with focus in Type hierarchy
	maximal expansion level in Call Hierarchy
	recursive call in Call Hierarchy

Build path



	class path variable
	unresolved class path variable
	JAR with attached source
	JAR without attached source
	system library
	reference to unavailable project
	reference to unavailable source folder
	reference to unavailable JAR
	build path ordering
	inclusion filter

	exclusion filter
	output folder
	Javadoc location
	source attachment












Code assist





















	HTML tag
	Javadoc tag
	local variable
	template











Compare

	field
	method







Debugger

	debug launch
	run launch
	terminated run launch
	process
	terminated process
	debug target
	suspended debug target
	terminated debug target
	thread
	suspended thread
	stack frame




	running stack frame
	adornment that marks a stack frame that may be out of synch with the target VM as a result of an unsuccessful hot code replace
	adornment that marks a stack frame that is out of synch with the target VM as a result of an unsuccessful hot code replace
	inspected object or primitive value
	watch expression
	local variable
	monitor
	a monitor in contention
	a thread in contention for a monitor
	a monitor that is owned by a thread
	a thread that owns a monitor
	current instruction pointer (top of stack)
	current instruction pointer
	enabled line breakpoint
	disabled line breakpoint
	adornment that marks a line breakpoints as installed
	adornment that marks a breakpoint as conditional
	adornment that marks an entry method breakpoint
	adornment that marks an exit method breakpoint
	field access watchpoint








	field modification watchpoint
	field access and modification watchpoint
	adornment that marks a watchpoint as installed
	exception breakpoint
	runtime exception breakpoint
	disabled exception breakpoint
	adornment that marks an exception breakpoint as caught
	adornment that marks an exception breakpoint as uncaught
	adornment that marks an exception breakpoint as scoped
	adornment that marks an exception breakpoint as installed

Editor





	implements
	overrides
	quick assist available
	search match
	collapsed
	expanded

JUnit











	test
	currently running test
	successful test



	failing test
	test throwing an exception
	test suite
	currently running test suite
	successfully completed test suite
	test suite with failing test
	test suite with exception throwing test

NLS tools










	skipped NLS key
	translated NLS key
	untranslated NLS key
	search for unused NLS keys

Quick fix





	quick fixable error
	quick fixable warning
	error that got fixed in source but file still needs a recompile
	warning that got fixed in source but file still needs a recompile
	add
	change
	change cast
	move to another package
	remove
	remove import

	rename
	surround with try/catch




Refactoring

	general change
	composite change
	compilation unit change
	text change
	file change
	Stop error
	Error
	Warning
	Information


Search


	Java Search
	search for declarations
	search for references
	search for unused NLS keys

Search – Occurrences in File

	a general match
	read access to local or field
	write access to local or field

Type hierarchy view

	type from non selected package
---	--------------------------------

	interface from non selected package
---	-------------------------------------

Dialog based refactoring user interface

A dialog based user interface guides you through the steps necessary to execute a selected refactoring. A dialog based refactoring user interface consists of a short first dialog gathering information that is required to execute the refactoring, a separate problem dialog that pops up if any errors are detected and a preview dialog to preview the results of a refactoring.

Input dialog

This dialog gathers information that is required for the refactoring. For example, for a rename refactoring you will enter the new name for the Java element. You can either press **OK** to execute the refactoring or **Preview >** to preview the result of the refactoring.

Preview dialog

The JDT allows you to preview the results of a refactoring action before you execute it.

The preview dialog consists of two parts:

- A tree at the top containing all Java elements affected by the refactoring. Each top-level node in the tree represents one compilation unit.
- A compare viewer at the bottom. The left side of the compare viewer shows the original, the right side displays the refactored source.

Problem dialog

The problem dialog indicates if there are suspected, potential, or definite problems with the refactoring action you are attempting.

Four types of problems are possible:

Information

A problem described as Information will not affect the refactoring in any way, nor will it negatively affect the code in the workbench. You can most likely ignore this type of problem.

Warnings

Warnings attempt to predict compiler warnings. This type of problem most likely will not negatively affect the code in your workbench.

Errors

A problem described as an Error is very likely to cause compiler errors or change your workbench code semantically. You can choose to continue with the refactoring in spite of these errors, although it is not recommended.

Stop problems

This type of problem prevents the refactoring from taking place. For example, if you select a comment and choose the Extract Method command from it, the workbench will issue a stop problem on the refactoring attempt because you cannot extract a comment.

If there aren't any stop problems then the refactoring can be carried out by pressing the **OK** button. To preview the results of the refactoring action, press the **Continue** button.

■ Related concepts

[Refactoring support](#)

■ Related tasks

[Refactoring without preview](#)

[Refactoring with preview](#)

[Previewing refactoring changes](#)

[Undoing a refactoring operation](#)

[Redoing a refactoring operation](#)

■ Related reference

[Refactoring actions](#)

[Icons](#)

Override methods

This dialog lets you define methods to override.

Use **Override/Implement Methods** from the Source menu or the context menu on a selected type or on a text selection in a type.

The dialog presents all methods that can be overridden from supertypes or implemented from interfaces. Abstract methods or unimplemented methods are selected by default.

The tree view groups methods by the type declaring the method. If more than one type in the hierarchy declare the same method, the method is only shown once, grouped to the first type in the list of supertypes that implements or defines this method.

The flat view shows only methods, sorted alphabetically.

When pressing **OK**, method stubs for all selected methods are created.

Option	Description	Default
Select methods to override or implement	Select methods to override or implement	Abstract methods from superclasses and unimplemented methods from interfaces are selected
Group methods by types	Shows methods grouped by a list of the super types in which they are declared.	selected
Select All	Select all methods	n/a
Deselect All	Deselect all methods	n/a

You can control whether Javadoc comments are added to the created methods with the **Generate method comments** option at the bottom of the dialog.

■ Related reference

Source actions

Extract method errors

When you attempt to extract a method, you may get one or more of the following common errors:

- Selected block references a local type declared outside the selection A local type declaration is not part of the selection but is referenced by one of the statements selected for extraction. Either extend the selection that it includes the local type declaration or reduce the selection that no reference to the local type declaration is selected.
- A local type declared in the selected block is referenced outside the selection The selection covers a local type declaration but the type is also referenced outside the selected statements. Either extend the selection that includes all references to the local type or reduce the selection that the local type declaration isn't selected.
- Ambiguous return value: selected block contains more than one assignment to local variable More than one assignment to a local variable was found inside the selected block. Either reduce the selection that only one assignment is selected or extend the selection that at least all reference except of one to the local variables are covered by the selection too.
- Ambiguous return value: expression access to local and return statement selected The selected statement generates more than one return value. This is for example the case if an expression is selected and an expression's argument is modified as well. To remedy this problem extend the selection to cover the read access of the modified argument as well.
- Selection contains a break statement but the corresponding break target isn't selected To remedy the problem either extend the selection to include the break / continue target or reduce the selection that the break / continue statement isn't covered by the selection.
- Selection contains a continue statement but the corresponding continue target isn't selected To remedy the problem either extend the selection to include the break / continue target or reduce the selection that the break / continue statement isn't covered by the selection.
- Selection starts inside a comment Parts of a comment cannot be extracted. Either extend the selection that it covers the whole comment or reduce the selection that the comment isn't covered at all.
- Selection ends inside a comment Parts of a comment can't be extracted. Either extend the selection that it covers the whole comment or reduce the selection that the comment isn't covered at all.
- Cannot extract selection that ends in the middle of a statement Adjust selection so that it fully covers a set of statements or expressions. The users can extend the selection to a valid range using the *Expand Selection to* in the Edit menu.

■ Related concepts

[Java development tools \(JDT\)](#)
[Refactoring support](#)

■ Related tasks

[Extracting a method](#)
[Using Structured Selection](#)

■ Related reference

Source menu

Refactor Menu

Extracting a local variable

To extract a local variable from an expression:

- In a Java editor, select the expression that you want to extract to a local variable
- Do one of the following:
 - ◆ From the editor's pop-up menu, select ***Refactor > Extract Local Variable*** or
 - ◆ From the menu bar, select ***Refactor > Extract Local Variable***

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Inlining a local variable

To inline a local variable:

- In a Java editor, select the variable that you want to inline (you can select a reference to the variable)
- Do one of the following:
 - ◆ From the menu bar, select ***Refactor > Inline*** or
 - ◆ From the editor's pop-up menu, select ***Refactor > Inline***

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Replacing a local variable with a query

To replace a local variable with a query:

- In the Java editor, select the expression with which the local variable is initialized
- Invoke the Extract Method refactoring by either:
 - ◆ Selecting **Refactor > Extract Method** from the editor's pop-up menu or
 - ◆ Selecting **Refactor > Extract Method** from the menu bar
- Perform the Extract Method refactoring
- Select the local variable (or a reference to it)
- Invoke the Inline Local Variable by either:
 - ◆ Selecting **Refactor > Inline** from the editor's pop-up menu or
 - ◆ Selecting **Refactor > Inline** from the menu bar
- Perform the Inline Local Variable refactoring

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Extracting a method](#)

[Inlining a local variable](#)

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Copying and moving Java elements

To move Java elements:

1. From a Java view, select the Java elements you want to move.
2. From the menu bar, select **Refactor > Move** or, from the view's pop-up menu, select **Refactor > Move**.
3. In the resulting dialog, select the new location for the element and press **OK**.
Note: Moving static members (such as methods and types), classes or compilation units allows you to choose to also update references to these elements. Additional options may be available; for more on these options, see the documentation for the corresponding "Rename" refactoring.

You can also move Java elements by dragging them and dropping in the desired new location.

Note: Dragging and dropping compilation units and types allows you to update references to these elements. In the dialog that appears on dropping, press **Yes** if you want to update references, press **Preview** if you want to see the preview of the reference updates, press **No** if you want to move the elements without updating references or press **Cancel** if you want to cancel the move operation. Additional options may be available; for more on these options, see the documentation for the corresponding "Rename" refactoring.

To copy Java elements you need to copy them to the clipboard and paste them in the desired new location:

1. From a Java view, select the Java elements you want to copy to the clipboard and do one of the following:
 - ◆ Press **Ctrl+C**
 - ◆ From the menu bar, select **Edit > Copy**
 - ◆ From the view's pop-up menu, select **Copy**
2. Now, to paste the elements, select the desired destination and do one of the following:
 - ◆ Press **Ctrl+V**
 - ◆ From the menu bar, select **Edit > Paste**
 - ◆ From the view's pop-up menu, select **Copy**

You can also copy Java elements by dragging them and dropping in the desired new location. You will need to have **Ctrl** pressed while dragging to copy the elements.

■ Related concepts

[Java projects](#)

■ Related tasks

[Creating Java elements](#)

[Moving folders, packages, and files](#)

[Copying and moving Java elements](#)

■ Related reference

[Edit menu](#)

Refactoring actions

Extracting a constant

To extract a constant from an expression:

- In a Java editor, select the expression that you want to extract to a constant
- Do one of the following:
 - ◆ From the editor's pop-up menu, select ***Refactor > Extract Constant*** or
 - ◆ From the menu bar, select ***Refactor > Extract Constant***

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Renaming a package

To rename a package:

1. In the Package Explorer or the Packages view select the package that you want to rename.
2. From the view's pop-up menu, select **Refactor > Rename**.

This updates all import statements of any affected compilation units and all fully qualified references to types declared in the renamed package.

■ Related concepts

[Java projects](#)

[Refactoring support](#)

■ Related tasks

[Opening a package](#)

[Moving folders, packages, and files](#)

[Creating a new Java package](#)

■ Related reference

[Package Explorer](#)

[Refactoring actions](#)

[Refactoring dialog](#)

[Java preferences](#)

Opening a package

To reveal a package in the tree of the Package Explorer:

1. Select *Navigate > Go To > Package* from the menu bar. The Go to Package dialog opens.
2. Type a package name in the *Choose a package* field to narrow the list of available packages, using wildcards as needed. As you type, the list is filtered to display only packages that match the current expression.
3. Select a package from the list, then click **OK**. The selected package is displayed in the Package Explorer.

■ Related concepts

Java views

■ Related tasks

Showing a type's compilation unit in the Package Explorer

Renaming a package

Moving folders, packages, and files

Creating a new Java package

■ Related reference

Navigate actions

Package Explorer

Showing an element in the Package Explorer view

You can reveal an element's location in the Package Explorer view

- 1. Select a Java element or activate a Java editor.
 2. From the menu bar, select *Navigate > Show In > Package Explorer*. If the Package Explorer is not already open, then it opens in the current perspective. The workbench navigates to the selected element (or the edited compilation unit).
- From the Java editor's pop-up menu, select *Show in Package Explorer*. The currently edited compilation unit will be revealed.

Note: The element might not be revealed if Package Explorer filters are active or the *Window > Preferences > Java > Appearance > Show Members in Package Explorer* preference is cleared.

■ Related concepts

[Java views](#)

■ Related tasks

[Setting execution arguments](#)

[Renaming a compilation unit](#)

[Opening a type in the Package Explorer](#)

[Organizing existing import statements](#)

■ Related reference

[Java Base preference page](#)

[Package Explorer](#)

Renaming a compilation unit

To rename a compilation unit:

1. In the Package Explorer, select the compilation unit you want to rename.
2. From the view's pop-up menu, select **Refactor** > **Rename**.

Renaming a compilation unit also renames (and updates all references to) the top-level type that has the same name as the compilation unit. For example, renaming a compilation unit *A.java* in which a class *A* is declared also renames class *A* and updates all references to it.

■ Related concepts

[Refactoring support](#)

■ Related tasks

[Copying and moving Java elements](#)

[Viewing compilation errors and warnings](#)

[Creating a class in an existing compilation unit](#)

[Creating a new interface in a compilation unit](#)

[Showing a type's compilation unit in the Packages view](#)

■ Related reference

[Package Explorer](#)

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Creating a new interface in an existing compilation unit

An alternative way to create a new interface is to add it to an existing compilation unit.

1. In the Package Explorer, double-click a compilation unit.
2. Type the code for the interface at the desired position in the compilation unit.

■ Related concepts

[Java projects](#)

■ Related tasks

[Creating Java elements](#)

[Creating a new Java interface](#)

[Creating a nested interface](#)

[Creating a top-level interface](#)

[Renaming a compilation unit](#)

■ Related reference

[Package Explorer](#)

Creating a new Java interface

Use the New Java Interface wizard to create a new Java interface. There are a number of ways to open this wizard:

1. Select the container where you want the new class to reside.
2. From the drop-down menu on the **New Java Class** button in the workbench toolbar, select **Interface**.

or

1. Select the container where you want the new class to reside.
2. From the container's pop-up menu, select **New > Interface**.

or

1. Select the container where you want the new class to reside.
2. From the drop-down menu on the **New** button in the workbench toolbar, select **Interface**.

or

1. Click the **New** button in the workbench toolbar to open the **New** wizard.
2. Select **Interface** or **Java > Interface** and click **Next**.

or

1. Select the container where you want the new interface to reside.
2. Then, select from the menu bar **File > New > Interface**.

■ Related concepts

[Java projects](#)

■ Related tasks

[Creating a top-level interface](#)

[Creating a nested interface](#)

[Creating a new interface in an existing compilation unit](#)

[Renaming a type](#)

■ Related reference

[New Java Interface wizard](#)

[Java Toolbar actions](#)

[Package Explorer](#)

Creating a top-level interface

You can create interfaces that are not enclosed in other types.

1. Open the New Java Interface wizard.
2. Edit the **Source Folder** field as needed to indicate in which folder you want the new interface to reside. You can either type a path or click the **Browse** button to find the folder. If a folder is found for the current selection, that folder appears in the **Source Folder** field as the container for the new interface.
3. In the **Package** field, type a name or click **Browse** to select the package where you want the new interface to reside. If you want the new interface to be created in the default package, leave this field empty.
4. Clear the **Enclosing type** checkbox.
5. In the **Name** field, type a name for the new interface. (Optionally, in a 5.0 project, add type parameters enclosed in < and >).
6. Select the **public** or **default** access modifier using the **Modifiers** radio buttons.
7. Click the **Add** button to add interfaces for the new interface to extend. (Optionally, in a 5.0 project, add type arguments enclosed in < and >).
8. Click **Finish**.

■ Related concepts

[Java projects](#)

■ Related tasks

[Creating a new Java interface](#)

[Creating a nested interface](#)

[Creating a new interface in a compilation unit](#)

[Renaming a type](#)

■ Related reference

[New Java Interface wizard](#)

[Java Toolbar actions](#)

[Package Explorer](#)

Creating a nested interface

You can create interfaces that are enclosed in other types (that is, nested interfaces).

1. Open the New Java Interface wizard.
2. Edit the **Source Folder** field to indicate in which folder you want the new interface to reside. You can either type a path or click the **Browse** button to find the folder. If a folder is found for the current selection, that folder appears in the **Source Folder** field as the container for the new interface.
3. Select the **Enclosing type** checkbox.
4. In the **Enclosing type** field, type the name of the enclosing type or click the **Browse** button to select the enclosing type for the new interface.
5. In the **Name** field, type a name for the new interface. (Optionally, in a 5.0 project, add type parameters enclosed in < and >).
6. Select the **public** or **default** access modifier by using the **Modifiers** radio buttons.
7. Select the **static** checkbox if you want the new interface to be static.
8. Click the **Add** button to add interfaces for the new interface to extend. (Optionally, in a 5.0 project, add type arguments enclosed in < and >).
9. Click **Finish** when you are done.

Note: The new interface is created in the same compilation unit as its enclosing type.

■ Related concepts

[Java projects](#)

■ Related tasks

[Creating a new Java interface](#)

[Creating a top-level interface](#)

[Creating a new interface in a compilation unit](#)

[Renaming a type](#)

■ Related reference

[New Java Interface wizard](#)

[Java Toolbar actions](#)

[Package Explorer](#)

Renaming a type

You can rename a type by modifying its declaration in the compilation unit in which it is declared. However, if you also want to update all references to it, do one of the following:

1. In a Java view presenting types (e.g. the Outline view, the Type Hierarchy views, etc.) select a type.
2. From the type's pop-up menu, select **Refactor > Rename** or use the **Refactor > Rename** action from the global menu bar.

or

1. In a Java editor, select a reference to the type.
2. From the editor's pop-up menu, select **Refactor > Rename** or use the **Refactor > Rename** action from the global menu bar.

■ Related concepts

[Java projects](#)

■ Related tasks

[Creating a new Java class](#)

[Creating a new Java enum](#)

[Creating a new Java annotation](#)

[Creating a nested interface](#)

[Creating a top-level interface](#)

[Creating a top-level class](#)

[Creating a nested class](#)

[Creating a class in an existing compilation unit](#)

■ Related reference

[Package Explorer](#)

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Creating a new Java enum

Use the New Java Enum wizard to create a new Java enum. There are a number of ways to open this wizard:

1. Select the container where you want the new enum to reside.
2. Click the **New Java Enum** button in the workbench toolbar.

or

1. Select the container where you want the new enum to reside.
2. From the container's pop-up menu, select **New > Enum**.

or

1. Select the container where you want the new enum to reside.
2. From the drop-down menu on the **New** button in the workbench toolbar, select **Enum**.

or

1. Click the **New** button in the workbench toolbar to open the **New** wizard.
2. Select **Java > Enum** and click **Next**.

or

1. Select the container where you want the new enum to reside.
2. Then, select from the menu bar **File > New > Enum**.

■ Related concepts

[Java projects](#)

■ Related tasks

[Renaming a type](#)

■ Related reference

[New Java Enum wizard](#)

[Java Toolbar actions](#)

[Package Explorer](#)

New Java Enum Wizard

This wizard helps you to create a new Java enum in in a Java project.

Java Enum Options

Option	Description	Default
Source folder	Enter a source folder for the new enum. Either type a valid source folder path or click Browse to select a source folder via a dialog.	The source folder of the element that was selected when the wizard has been started.
Package	Enter a package to contain the new enum. You can select either this option or the Enclosing Type option, below. Either type a valid package name or click Browse to select a package via a dialog.	The package of the element that was selected when the wizard has been started.
Enclosing type	Select this option to choose a type in which to enclose the new enum. You can select either this option or the Package option, above. Either type a valid name in the field or click Browse to select a type via a dialog.	The type or the primary type of the compilation unit that was selected when the wizard has been started or <blank>
Name	Type a name for the new enum.	<blank>
Modifiers	Select one or more access modifiers for the new enum. <ul style="list-style-type: none">• Either public, default, private, or protected (private and protected are only available if you specify an enclosing type)	public
Interfaces	Click Add to choose interfaces that the new enum implements.	<blank>
Do you want to add comments?	When selected, the wizard adds comments to the new enum where appropriate.	Do not add comments

■ Related tasks

Creating a new Java enum

■ Related reference

File actions

Creating a new Java annotation

Use the New Java Annotation wizard to create a new Java annotation. There are a number of ways to open this wizard:

1. Select the container where you want the new class to reside.
2. From the drop-down menu on the **New Java Class** button in the workbench toolbar, select **Annotation**.

or

1. Select the container where you want the new class to reside.
2. From the container's pop-up menu, select **New > Annotation**.

or

1. Select the container where you want the new class to reside.
2. From the drop-down menu on the **New** button in the workbench toolbar, select **Annotation**.

or

1. Click the **New** button in the workbench toolbar to open the **New** wizard.
2. Select **Java > Annotation** and click **Next**.

or

1. Select the container where you want the new annotation to reside.
2. Then, select from the menu bar **File > New > Annotation**.

■ Related concepts

[Java projects](#)

■ Related tasks

[Renaming a type](#)

■ Related reference

[New Java Annotation wizard](#)

[Java Toolbar actions](#)

[Package Explorer](#)

New Java Annotation Wizard

This wizard helps you to create a new Java Annotation in a Java project.

Java Annotation Options

Option	Description	Default
Source folder	Enter a source folder for the new annotation. Either type a valid source folder path or click Browse to select a source folder via a dialog.	The source folder of the element that was selected when the wizard has been started.
Package	Enter a package to contain the new annotation. You can select either this option or the Enclosing Type option, below. Either type a valid package name or click Browse to select a package via a dialog.	The package of the element that was selected when the wizard has been started.
Enclosing type	Select this option to choose a type in which to enclose the new annotation. You can select either this option or the Package option, above. Either type a valid name in the field or click Browse to select a type via a dialog.	The type or the primary type of the compilation unit that was selected when the wizard has been started or <blank>
Name	Type a name for the new annotation.	<blank>
Modifiers	Select one or more access modifiers for the new annotation. <ul style="list-style-type: none">• Either public, default, private, or protected (private and protected are only available if you specify an enclosing type)	public
Do you want to add comments?	When selected, the wizard adds comments to the new class where appropriate.	Do not add comments

Related tasks

[Creating a new Java annotation](#)

Related reference

[File actions](#)

Creating a top-level class

You can create classes that are not enclosed in other types.

1. Open the New Class wizard.
2. Edit the **Source Folder** field as needed to indicate in which folder you want the new class to reside. You can either type a path or click the **Browse** button to find the folder. If a folder is found for the current selection, that folder appears in the **Source Folder** field as the container for the new class.
3. In the **Package** field, type or click **Browse** to select the package where you want the new class to reside. Leave this field empty to indicate that you want the new class to be created in the default package.
4. Leave the **Enclosing type** box deselected.
5. In the **Name** field, type a name for the new class. (Optionally, in a 5.0 project, add type parameters enclosed in < and >).
6. Select the **public** or **default** access modifier using the **Modifiers** radio buttons.
Note: The private and protected options are available only when creating a class enclosed in a type.
7. Optionally, select the **abstract** or **final** modifier for the new class using the appropriate checkboxes:
Note: The static option is available only when creating a class enclosed in a type.
8. In the **Superclass** field, type or click **Browse** to select a superclass for the new class. (Optionally, in a 5.0 project, add type arguments enclosed in < and >).
9. Click the **Add** button to add interfaces for the new class to implement. (Optionally, in a 5.0 project, add type arguments enclosed in < and >).
10. If you want to create some method stubs in the new class:
 - ◆ Select the **public static void main(String[] args)** box if you want the wizard to add the main method to the new class, thus making it a starting point of an application.
 - ◆ Select the **Constructors from superclass** checkbox if you want the wizard to create, in the new class, a set of constructors, one for each of the constructors declared in the superclass. Each of them will have the same number of parameters (of the same types) as the respective constructor from the superclass.
 - ◆ Select the **Inherited abstract methods** checkbox if you want the wizard to generate method stubs for each of the abstract methods that the new class will inherit from its superclass and implemented interfaces.
11. Click **Finish** when you are done.

■ Related concepts

[Java projects](#)

■ Related tasks

[Creating a new Java class](#)

[Creating a nested class](#)

[Creating a class in an existing compilation unit](#)

■ Related reference

[New Java Project wizard](#)

[New Source Folder wizard](#)

New Java Package wizard

New Java Class wizard

Java Toolbar actions

Creating a nested class

You can create classes that are enclosed in other types (that is, nested classes).

1. Open the New Java Class wizard.
2. Edit the **Source Folder** field to indicate in which folder you want the new class to reside. You can either type a path or click **Browse** to find the folder. If a folder is found for the current selection, that folder appears in the **Source Folder** field as the container for the new class.
3. Select the **Enclosing Type** checkbox and type the name of the enclosing type in the **Enclosing Type** field. You can also click **Browse** to select the enclosing type for the new class.
4. In the **Name** field, type a name for the new class. (Optionally, in a 5.0 project, add type parameters enclosed in < and >).
5. Select the desired modifiers by using the **Modifiers** radio buttons and checkboxes.
6. In the **Superclass** field, type or click **Browse** to select a superclass for the new class. (Optionally, in a 5.0 project, add type arguments enclosed in < and >).
7. Click the **Add** button to add interfaces for the new class to implement. (Optionally, in a 5.0 project, add type arguments enclosed in < and >).
8. If you want to create some method stubs in the new class:
9.
 - ◆ Select the **public static void main(String[] args)** checkbox if you want the wizard to add the main method to the new class, thus making it a starting point of an application.
 - ◆ Select the **Constructors from superclass** checkbox if you want the wizard to create, in the new class, a set of constructors, one for each of the constructors declared in the superclass. Each of them will have the same number of parameters (of the same types) as the respective constructor from the superclass.
 - ◆ Select the **Inherited abstract methods** checkbox if you want the wizard to generate method stubs for each of the abstract methods that the new class will inherit from its superclass and implemented interfaces.
10. Click **Finish** when you are done.

Note: The new class is created in the same compilation unit as its enclosing type.

■ Related concepts

[Java projects](#)

■ Related tasks

[Creating a new Java class](#)

[Creating a top-level class](#)

[Creating a class in an existing compilation unit](#)

■ Related reference

[New Java Class wizard](#)

New Java Class Wizard

This wizard helps you to create a new Java class in in a Java project.

Java Class Options

Option	Description	Default
Source folder	Enter a source folder for the new class. Either type a valid source folder path or click Browse to select a source folder via a dialog.	The source folder of the element that was selected when the wizard has been started.
Package	Enter a package to contain the new class. You can select either this option or the Enclosing Type option, below. Either type a valid package name or click Browse to select a package via a dialog.	The package of the element that was selected when the wizard has been started.
Enclosing type	Select this option to choose a type in which to enclose the new class. You can select either this option or the Package option, above. Either type a valid name in the field or click Browse to select a type via a dialog.	The type or the primary type of the compilation unit that was selected when the wizard has been started or <blank>
Name	Type a name for the new class.	<blank>
Modifiers	Select one or more access modifiers for the new class. <ul style="list-style-type: none">• Either public, default, private, or protected (private and protected are only available if you specify an enclosing type)• abstract• final• static (only available if you specify an enclosing type)	public
Superclass	Type or click Browse to select a superclass for this class.	The type (not the compilation unit!) that was selected when the wizard has been started or <java.lang.Object>
Interfaces	Click Add to choose interfaces that the new class implements.	<blank>
Which method stubs would you like to create?	Choose the method stubs to create in this class: <ul style="list-style-type: none">• public static void main(String [] args): Adds a main method stub to the new class.	Inherited abstract methods enabled

Basic tutorial

	<ul style="list-style-type: none">• Constructors from superclass: Copies the constructors from the new class's superclass and adds these stubs to the new class.• Inherited abstract methods: Adds to the new class stubs of any abstract methods from superclasses or methods of interfaces that need to be implemented.	
Do you want to add comments?	When selected, the wizard adds comments to the new class where appropriate.	Do not add comments

Related tasks

Creating a new Java class

Related reference

File actions

New Source Folder Wizard

This wizard helps you to create a new source folder to a Java project.

Note that a new source folder can not be nested in existing source folders or in an output folder. You can choose to add exclusion filters to the other nesting source folders or the wizard will suggest to replace the nesting classpath entry with the new created entry. The wizard will also suggest to change the output location.

New Source Folder Options

Option	Description	Default
Project name	Enter a project to contain the new source folder. Either type a valid project name or click Browse to select a project via a dialog.	The project of the element that was selected when the wizard has been started.
Folder name	Type a path for the new source folder. The path is relative to the selected project.	<blank>
Update exclusion filter in other source folders to solve nesting	Select to modify existing source folder's exclusion filters to solve nesting problems. For example if there is an existing source folder <i>src</i> and a folder <i>src/inner</i> is created, the source folder <i>src</i> will be updated to have a exclusion filter <i>src/inner</i> .	Off

■ Related concepts

[Java projects](#)

■ Related tasks

[Creating Java elements](#)

[Creating a new source folder](#)

■ Related reference

[File actions](#)

New Java Interface Wizard

This wizard helps you to create a new Java interface in a Java project.

Java Interface Options

Option	Description	Default
Source folder	Enter a source folder for the new interface. Either type a valid source folder path or click Browse to select a source folder via a dialog.	The source folder of the element that was selected when the wizard has been started.
Package	Enter a package to contain the new interface. You can select either this option or the Enclosing Type option, below. Either type a valid package name or click Browse to select a package via a dialog.	The package of the element that was selected when the wizard has been started.
Enclosing type	Select this option to choose a type in which to enclose the new interface. You can select either this option or the Package option, above. Either type a valid name in the field or click Browse to select a type via a dialog.	The type or the primary type of the compilation unit that was selected when the wizard has been started or <blank>
Name	Type a name for the new interface.	<blank>
Modifiers	Select one or more access modifiers for the new interface. <ul style="list-style-type: none">• Either public, default, private, or protected (private and protected are only available if you specify an enclosing type)• static (only available if you specify an enclosing type)	public
Extended interfaces	Click Add to choose interfaces that the new interface extends.	<blank>
Do you want to add comments?	When selected, the wizard adds comments to the new class where appropriate.	Do not add comments

Related tasks

[Creating a new Java interface](#)

Related reference

[File actions](#)

Opening a type in the Package Explorer view

You can open the Package Explorer on any type that is included on a project's class path.

1. From the menu bar, select **Navigate > Go To > Type**. The Go to Type dialog opens.
2. In the **Choose a type** field, begin typing an expression to narrow the list of available types, using wildcards as needed. As you type, the list is filtered to display only types that match the current expression.
3. In the **Matching types** list, select a type. Hint: you can press the **Down** key to move to the first type.
4. Click **OK** when you are done. The selected type is displayed in the Package Explorer.

Note: The Goto Type dialog maintains a history of recently opened types. These are shown when the dialog is opened and stay above a separator line when you start to type a filter expression.

Note: Revealing may not be possible if Package Explorer filters are applied.

■ Related concepts

Java development tools (JDT)

■ Related tasks

Showing a type's compilation unit in the Package Explorer

■ Related reference

Navigate actions

Package Explorer

Organizing existing import statements

The Java editor can help you improve the existing import statements inside a compilation unit.

1. Do one of the following while editing your Java code:
 - ◆ Select **Source > Organize Imports** from the pop-up menu in the editor
 - ◆ Select **Source > Organize Imports** from the menu bar
 - ◆ Press **Ctrl+Shift+O**
2. The Java editor generates a complete list of import statements, as specified by the import order preference, and new import statements replace the old ones.

Note: **Source > Organize Imports** also works on whole packages or projects – just select them in the Package Explorer.

■ Related concepts

[Java editor](#)

■ Related tasks

[Adding required import statements](#)

[Managing import statements](#)

[Setting the order of import statements](#)

[Showing a type's compilation unit in the Package Explorer view](#)

■ Related reference

[Source menu](#)

Adding required import statements

The Java editor can help you adding required import statements for a selected type inside a compilation unit.

1. Select a reference to a type in your Java code, and do one of the following:
 - ◆ Select **Source > Add Import** from the pop-up menu in the editor
 - ◆ Select **Source > Add Import** from the menu bar.
 - ◆ Press **Ctrl + Shift + M**
2. Either the editor can identify the type or you are prompted to choose the desired type from a list of possible types.
3. The import statement is generated and inserted as specified by the import order preference.

■ Related concepts

Java editor

■ Related tasks

Using the Java editor

Managing import statements

Organizing existing import statements

Setting the order of import statements

■ Related reference

Source menu

Managing import statements

The default Java editor includes several features that help you manage import statements.

■ Related concepts

[Java editor](#)

■ Related tasks

[Using the Java editor](#)

[Adding required import statements](#)

[Organizing existing import statements](#)

[Setting the order of import statements](#)

■ Related reference

[Source menu](#)

Setting the order of import statements

1. From the menu bar, select **Window > Preferences**.
2. In the left pane, expand the **Java > Code Style** category and select **Organize Imports**.
3. The Organize Imports page defines the sorting order of import statements. In the Imports list, manage the list of package prefixes as follows:
 - ◆ **New** to add a new prefix
 - ◆ **New Static** to add a new prefix for static imports (J2SE 5.0 only)
 - ◆ **Edit** to change the name of an existing prefix
 - ◆ Use **Up** and **Down** buttons to rearrange the sequence of the list by moving the selected prefix up or down
 - ◆ **Remove** to remove the selected prefix from the list
 - ◆ Use **Import...** and **Export...** to load a list of prefixes from a file or to store it to a file
4. In the **Number of imports needed before .*** field, type the number of import statements that are allowed to refer to the same package before `<package prefix>.*` is used. This number is called the import threshold.
5. Click **OK** when you are done.

Note: The order of import statements can also be configured per project:

1. Select a java project, open the pop-up menu and choose **Properties**.
2. Select the **Code Style > Organize Imports** page and check **Enable project specific settings**.
3. Manage the list as explained above.
4. Click **OK** when you are done.

■ Related concepts

[Java editor](#)

■ Related tasks

[Adding required import statements](#)

[Managing import statements](#)

[Organizing existing import statements](#)

■ Related reference

[Refactoring actions](#)

[Organize Import preference page](#)

Organize Imports

The following preferences define how the Organize Imports command generates the import statements in a compilation unit.

Organize Imports Preferences

Option	Description	Default
Import order list	This list of prefixes shows the sequential order for packages imported into a Java compilation unit. Each entry defines a block. Different blocks are separated by a spacer line.	java javax org com
New...	Adds a package name prefix to the import order list. In the resulting dialog, type a package name or package name prefix.	n/a
New Static...	Adds a package name prefix to the import order list. In the resulting dialog, type a package name or package name prefix.	n/a
Edit...	Change the name of an existing package name prefix. In the resulting dialog, type a package name or package name prefix.	n/a
Remove	Removes a package name prefix from the import order list.	n/a
Up	Moves the selected package name prefix up in the import order list.	n/a
Down	Moves the selected package name prefix down in the import order list.	n/a
Import...	Load a list of package name prefixes from a file.	n/a
Export...	Save the list of package name prefixes to a file.	n/a
Number of imports needed for .*	The number of fully-qualified import statements that are allowed from the same package before <i><package>.*</i> is used.	99
Do not create imports for types starting with a lowercase letter	If enabled, types starting with a lowercase letter are not imported.	On

Related tasks

Managing import statements

Related reference

Source actions

Renaming a field

You can rename a field by modifying its declaration in the compilation unit in which it is declared. However, if you also want to update all references to it, do one of the following:

1. In a Java view presenting fields (for example in the Outline view) select a field.
2. From the view's pop-up menu, select **Refactor > Rename** or select **Refactor > Rename** from the global menu bar.

or

1. In a Java editor, select a reference to the field (or the field's declaration).
2. From the editor's pop-up menu, select **Refactor > Rename** or select **Refactor > Rename** from the global menu bar.

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

[Package Explorer](#)

Renaming a local variable

To rename a local variable (or a method parameter):

- Select the variable (or a reference to it) in the Java editor
- Do one of the following:
 - ◆ From the menu bar, select Refactor > Rename or
 - ◆ From the editor's pop-up menu, select Refactor > Rename

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Parameters page](#)

[Extracting a local variable](#)

[Inlining a local variable](#)

[Renaming method parameters](#)

[Changing method signature](#)

[Replacing a local variable with a query](#)

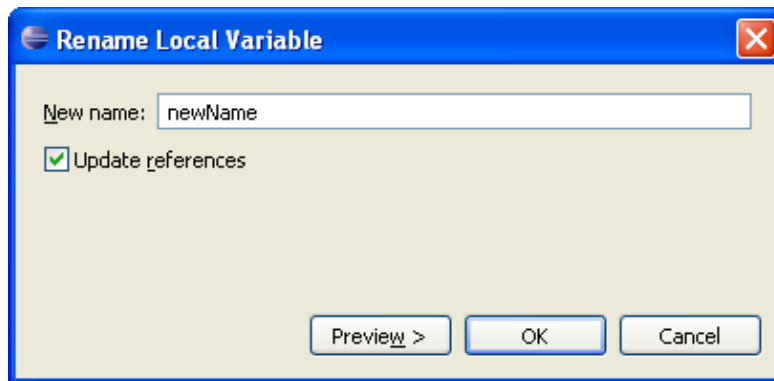
■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Parameters page



Parameters Page for the Rename Local Variable Refactoring Command

- In the Enter new name field, type a new name for the local variable.
- If you do not want to update references to the renamed local variable, deselect the Update references to the renamed element checkbox.
- Click OK to perform a quick refactoring, or click Preview to perform a controlled refactoring.

■ Related tasks

[Renaming a local variable](#)

[See Refactoring without Preview](#)

[See Refactoring with Preview](#)

Inlining a method

To inline a method:

- In a Java editor or in one of the Java views, select the method that you want to inline (you can also select an invocation site of the method)
- Do one of the following:
 - ◆ From the menu bar, select ***Refactor > Inline*** or
 - ◆ From the editor's pop-up menu, select ***Refactor > Inline***

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Inlining a constant

To inline a constant:

- In a Java editor or in one of the Java views, select the constant that you want to inline (you can select a reference to the constant)
- Do one of the following:
 - ◆ From the menu bar, select ***Refactor > Inline*** or
 - ◆ From the editor's pop-up menu, select ***Refactor > Inline***

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Self encapsulating a field

To self-encapsulate a field:

- Select the field in one of the Java views (e.g. Outline, Package Explorer or Members view)
- Do one of the following
 - ◆ From the menu bar, select ***Refactor > Self Encapsulate*** or
 - ◆ From the field's pop-up menu, select ***Refactor > Self Encapsulate***

You can also invoke this refactoring from the Java editor:

- In the Java editor, select the field (or a reference to it)
- Do one of the following
 - ◆ From the menu bar, select ***Refactor > Self Encapsulate*** or
 - ◆ From the editor's pop-up menu, select ***Refactor > Self Encapsulate***

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Pulling members up to superclass

To pull up class members (fields and methods) to the class's superclass:

- In a Java view (e.g. Outline, Package Explorer, Members), select the members that you want to pull up.
- Do one of the following:
 - ◆ From the menu bar, select **Refactor > Pull Up** or
 - ◆ From the pop-up menu, select **Refactor > Pull Up**

Note: the selected members must all have the same declaring type for this refactoring to be enabled.

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Pushing members down to subclasses

To push down class members (fields and methods) to the class's subclasses:

- In a Java view (e.g. Outline, Package Explorer, Members), select the members that you want to push down.
- Do one of the following:
 - ◆ From the menu bar, select **Refactor > Push Down** or
 - ◆ From the pop-up menu, select **Refactor > Push Down**

Note: the selected members must all have the same declaring type for this refactoring to be enabled.

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Moving static members between types

To move static members (fields and methods) between types:

- In a Java view, select the static members that you want to move
- Do one of the following:
 - ◆ From the menu bar, select ***Refactor > Move*** or
 - ◆ From the pop-up menu select, ***Refactor > Move***

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Moving an instance method to a component

To move an instance method to a component:

- In a Java view or in the Java editor, select the method that you want to move
- Do one of the following:
 - ◆ From the menu bar, select ***Refactor > Move*** or
 - ◆ From the pop-up menu select, ***Refactor > Move***

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Converting a local variable to a field

To convert a local variable to a field:

- In a Java editor or in one of the Java views, select the local variable
- Do one of the following:
 - ◆ From the menu bar, select ***Refactor > Convert Local Variable to Field*** or
 - ◆ From the editor's pop-up menu, select ***Refactor > Convert Local Variable to Field***

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Converting an anonymous inner class to a nested class

To convert an anonymous inner class to a nested class:

- In a Java editor, position the care inside the anonymous class
- Do one of the following:
 - ◆ From the menu bar, select ***Refactor > Convert Anonymous Class to Nested*** or
 - ◆ From the editor's pop-up menu, select ***Refactor > Convert Anonymous Class to Nested***

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Converting a nested type to a top level type

To convert a nested type to a top level type:

- In a Java editor or a Java view, select the member type
- Do one of the following:
 - ◆ From the menu bar, select ***Refactor > Convert Nested Type to Top Level*** or
 - ◆ From the editor's pop-up menu, select ***Refactor > Convert Nested Type to Top Level***

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Extracting an interface from a type

To extract an interface from a type:

- In a Java editor or a Java view, select the type from which you want to extract an interface
- Do one of the following:
 - ◆ From the menu bar, select ***Refactor > Extract Interface*** or
 - ◆ From the editor's pop-up menu, select ***Refactor > Extract Interface***

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Replacing references to a type with references to one of its supertypes

To replace references to a type with references to one of its supertypes:

- In a Java editor or a Java view, select the type
- Do one of the following:
 - ◆ From the menu bar, select *Refactor > Use Supertype Where Possible* or
 - ◆ From the editor's pop-up menu, select *Refactor > Use Supertype Where Possible*

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Replacing a single reference to a type with a reference to one of its supertypes

To replace a single reference to a type with a reference to one of its supertypes:

- In the Java editor, select the type reference, or the declaration of a field, parameter, or local variable
- Do one of the following:
 - ◆ From the menu bar, select **Refactor > Generalize Type** or
 - ◆ From the editor's pop-up menu, select **Refactor > Generalize Type**

You will be prompted for the type to which you would like to update the reference. If the reference can be safely changed to the new type, the refactoring proceeds.

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Replacing an expression with a method parameter

To replace an expression with a method parameter:

- In the Java editor, select the expression
- Do one of the following:
 - ◆ From the menu bar, select **Refactor > Introduce Parameter** or
 - ◆ From the editor's pop-up menu, select **Refactor > Introduce Parameter**

The highlighted expression will be replaced with a reference to a new method parameter. Callers of the method will be updated to pass the expression as the value of the new parameter.

Note: this refactoring may result in non-compiling code if the highlighted expression explicitly or implicitly references `this`.

Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Replacing constructor calls with factory method invocations

To replace calls to a constructor with calls to an equivalent factory method:

- In a Java editor or Java view, select the constructor declaration
- Do one of the following:
 - ◆ From the menu bar, select **Refactor > Introduce Factory** or
 - ◆ From the pop-up menu, select **Refactor > Introduce Factory**

You will be asked what to name the new factory method, on what class it should be placed, and whether to make the constructor private when the refactoring is complete. When the refactoring executes, it will create the new factory method, which will call the selected constructor and return the created object. All references to the constructor will be replaced by calls to the new factory method.

Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Inferring type parameters for generic type references

Eclipse can attempt to infer type parameters for all generic type references in a class, package, or project. This is especially useful when migrating from Java 1.4 code to Java 5.0 code, allowing you to automatically make use of the generic classes in Java's collections API.

- Open a class in the Java editor, or in a Java view, select a class, package, or project.
- Do one of the following:
 - ◆ From the menu bar, select **Refactor > Infer Generic Type Arguments** or
 - ◆ From the pop-up menu, select **Refactor > Infer Generic Type Arguments**

You will be given a dialog with two configurable options:

- **Assume clone() returns an instance of the receiver type** Well-behaved classes generally respect this rule, but if you know that your code violates it, uncheck the box.
- **Leave unconstrained type arguments raw (rather than inferring <?>)**. If there are no constraints on the elements of e.g. `ArrayList a`, unchecking this box will cause Eclipse to still provide a wildcard parameter, replacing the reference with `ArrayList<?> a`.

Press **OK** or **Preview** to continue with the operation.

Note: It may in some cases be impossible to assign consistent type parameters in a selection, or require deeper analysis than Eclipse can perform.

■ Related reference

[Refactoring actions](#)

[Refactoring dialogs](#)

[Java preferences](#)

Opening an editor on a type

You can open an editor on any type in the workbench.

1. Press **Ctrl+Shift+T** or, select **Navigate > Open Type** from the menu bar. The Open Type dialog opens.
2. In the **Choose a type** field, begin typing an expression to narrow the list of available types, using wildcards as needed. As you type, the list is filtered to display only types that match the current expression.
3. In the **Matching types** list, select a type. Hint: you can press the **Down** key to move to the first type.
4. Click **OK** when you are done. An editor opens on the selected type.

Note: The Open Type dialog maintains a history of recently opened types. These are shown when the dialog is opened and stay above a separator line when you start to type a filter expression.

Note: If you open a type from a CLASS or JAR file, you will see a special editor showing only method signatures unless you have attached source to it.

■ Related concepts

[Java editor](#)

■ Related tasks

[Attaching source to a JAR file](#)

[Opening an editor for a selected element](#)

[Using the Java editor](#)

■ Related reference

[Open Type dialog](#) [Navigate actions](#)

[Views and editors](#)

Open Type

This command allows you to browse the workbench for a type to open in an editor or type hierarchy

- **Select a type to open:** In this field, type the first few characters of the type you want to open in an editor. You may use wildcards as needed ("?" for any character, "*" for any string, and "TZ" for types containing "T" and "Z" as upper-case letters in camel-case notation, e.g. `java.util.TimeZone`).
- **Matching types:** This list displays matches for the expression you type in the *Select a type to open* field.

The behavior of the *Open Type* dialog can be further customized using the dialog menu:

Open Type Preferences

Option	Description	Default
Fully Qualify Duplicates	When selected, duplicate matches are displayed using fully qualified names, packages, and container information (e.g. the package and containing JRE)	Do not fully qualify
Show Container Info	When selected, the <i>Open Type</i> dialog shows an additional bar at the bottom of the dialog which displays the package and containing JRE of the selected type	Do not show container info

Additionally, the dialog menu allows to use working sets to further constrain the matching types.

■ Related tasks

[Opening an editor on a type](#)

■ Related reference

[Navigate actions](#)

Project actions

Project menu commands:

Name	Function	<i>Keyboard Shortcut</i>
Open Project	Shows a dialog that can be used to select a closed project and open it	
Close Project	Closes the currently selected projects	
Build All	Builds the all projects in the workspace. This is an incremental build, meaning that the builder analyzes the changes since the last time of build and minimizes the number of changed files.	Ctrl + B
Build Project	Builds the currently selected project. This is an incremental build, meaning that the builder analyzes the changes since the last time of build and minimizes the number of changed files.	
Build Working Set	Builds the projects contained in the currently selected working set. This is an incremental build, meaning that the builder analyzes the changes since the last time of build and minimizes the number of changed files.	
Clean...	Shows a dialog where the projects to be cleaned can be selected.	
Build Automatically	If selected, all modified files are automatically rebuilt if saved. This is an incremental build, meaning that the builder analyzes the changes since the last time of build and minimizes the number of changed files.	
Generate Javadoc...	Opens the Generate Javadoc wizard on the currently selected project.	
Properties	Opens the property pages on the currently selected project.	

■ Related concepts

[Java projects](#)

[Java builder](#)

■ Related tasks

[Building a Java program](#)

Run menu

This menu allows you to manage the running of an application in the workbench. Some menu items are only active if the Debug view is the active view.

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Launching a Java program](#)

[Running and debugging](#)

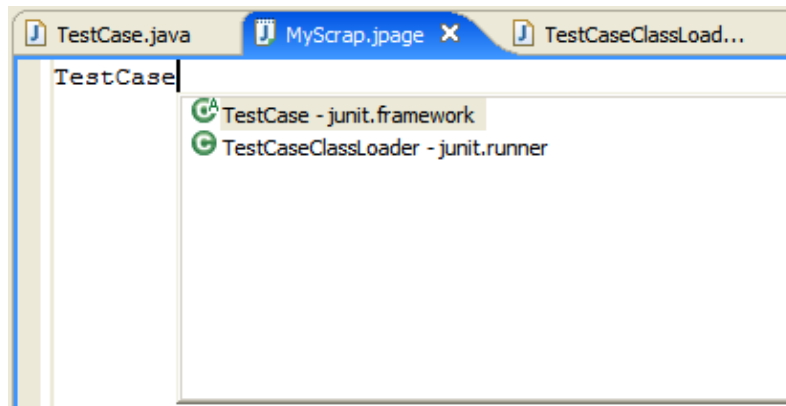
■ Related reference

[Run and debug actions](#)

Content/Code Assist

If activated from a valid line of code in an editor, this command opens a scrollable list of available code completions. Some tips for using code assist are listed in the following paragraph:

- If you select and then hover over a selected line in the content assist list, you can view Javadoc information for that line.
- You can use the mouse or the keyboard (Up Arrow, Down Arrow, Page Up, Page Down, Home, End, Enter) to navigate and select lines in the list.
- Clicking or pressing Enter on a selected line in the list inserts the selection into the editor.
- You can access specialized content assist features inside Javadoc comments.



Configure the behavior of the content assist in the [Java > Editor > Code Assist](#) preference page.

■ Related concepts

[Java editor](#)

[Java Development Tools \(JDT\)](#)

■ Related tasks

[Using content/code assist](#)

■ Related reference

[Edit menu](#)

[Java editor preferences](#)

[Templates preferences](#)

Templates

The Templates preference page allows to create new and edit existing templates. A template is a convenience for the programmer to quickly insert often reoccurring source code patterns.

The following buttons allow manipulation and configuration of templates:

Action	Description
New...	Opens a dialog to create a new template.
Edit...	Opens a dialog to edit the currently selected template.
Remove	Removes all selected templates.
Import...	Imports templates from the file system.
Export...	Exports all selected templates to the file system.
Export All...	Exports all templates to the file system.
Enable All	Enables all templates.
Disable All	Disables all templates.
Use code formatter	If enabled, the template is formatted according to the code formatting rules specified in the Code Formatter preferences , prior to insertion. Otherwise, the template is inserted as is, but correctly indented.

Template dialog

Creating a new template and editing an existing template uses the same dialog, which is described here.

The following fields and buttons appear in the dialog:

Option	Description
Name	The name of the template.
Context	The context determines where the template can be used and the set of available pre-defined template variables. Java The Java context Javadoc The Javadoc context
Automatically insert	If selected, code assist will automatically insert the template if it is the only proposal available at the caret position.
Description	A description of the template, which is displayed to the user when choosing the template.
Pattern	The template pattern.

Insert Variables...	Displays a list of pre-defined context specific variables.
---------------------	--

Template variables

Both Java and Javadoc context define the following variables:

Variable	Description
<i><code>\${cursor}</code></i>	Specifies the cursor position when the template edit mode is left. This is useful when the cursor should jump to another place than to the end of the template on leaving template edit mode.
<i><code>\${date}</code></i>	Evaluates to the current date.
<i><code>\${dollar}</code></i>	Evaluates to the dollar symbol '\$'. Alternatively, two dollars can be used: '\$\$'.
<i><code>\${enclosing_method}</code></i>	Evaluates to the name of the enclosing name.
<i><code>\${enclosing_method_arguments}</code></i>	Evaluates to a comma separated list of argument names of the enclosing method. This variable can be useful when generating log statements for many methods.
<i><code>\${enclosing_package}</code></i>	Evaluates to the name of the enclosing package.
<i><code>\${enclosing_project}</code></i>	Evaluates to the name of the enclosing project.
<i><code>\${enclosing_type}</code></i>	Evaluates to the name of the enclosing type.
<i><code>\${file}</code></i>	Evaluates to the name of the file.
<i><code>\${line_selection}</code></i>	Evaluates to content of all currently selected lines.
<i><code>\${primary_type_name}</code></i>	Evaluates to the name primary type of the current compilation unit.
<i><code>\${return_type}</code></i>	Evaluates to the return type of the enclosing method.
<i><code>\${time}</code></i>	Evaluates to the current time.
<i><code>\${user}</code></i>	Evaluates to the user name.
<i><code>\${word_selection}</code></i>	Evaluates to the content of the current text selection.
<i><code>\${year}</code></i>	Evaluates to the current year.

The Java context additionally defines the following variables:

Variable	Description
<i><code>\${array}</code></i>	Evaluates to a proposal for a declared array name.
<i><code>\${array_element}</code></i>	Evaluates to a proposal for an element name of a declared array.
<i><code>\${array_type}</code></i>	Evaluates to a proposal for the element type of a declared array.
<i><code>\${collection}</code></i>	

Basic tutorial

	Evaluates to a proposal for a declared collection implementing <code>java.util.Collection</code> .
<i><code>\${index}</code></i>	Evaluates to a proposal for an undeclared array index iterator.
<i><code>\${iterator}</code></i>	Evaluates to a proposal for an undeclared collection iterator.
<i><code>\${iterable}</code></i>	Evaluates to a proposal for a declared iterable name.
<i><code>\${iterable_element}</code></i>	Evaluates to a proposal for an element name of a declared iterable.
<i><code>\${iterable_type}</code></i>	Evaluates to a proposal for the element type of a declared iterable.
<i><code>\${todo}</code></i>	Evaluates to a proposal for the currently specified default task tag.

■ Related concepts

Templates

■ Related tasks

Using templates

Writing your own templates

■ Related reference

Java content assist

Task tag preferences

Code templates preferences

Code style preferences

Templates

Templates are a structured description of coding patterns that reoccur in source code. The Java editor supports the use of templates to fill in commonly used source patterns. Templates are inserted using content assist (*Ctrl+Space*).

For example, a common coding pattern is to iterate over the elements of an array using a for loop that indexes into the array. By using a template for this pattern, you can avoid typing in the complete code for the loop. Invoking content assist after typing the word `for` will present you with a list of possible templates for a for loop. You can choose the appropriate template by name (`iterate over array`). Selecting this template will insert the code into the editor and position your cursor so that you can edit the details.

Many common templates are already defined. These can be browsed in **Window > Preferences > Java > Editor > Templates**. You can also create your own templates or edit the existing ones.

■ Related tasks

[Using templates](#)

[Writing your own templates](#)

■ Related reference

[Edit menu](#)

[Java Content Assist](#)

[Templates preferences](#)

Using templates

To use templates:

1. In the Java editor, position the caret in a place where you want to insert a template.
2. Invoke content assist by pressing **Ctrl+Space**.
3. Templates appear in the presented list. Note that the list is filtered as you type, so typing a few first characters of a template name will reveal it.
4. Note that a preview is presented for each selected template.

Notes:

Templates can have variables, which are place-holders for the dynamic part of a template pattern, e.g. subject to change with every application of the particular template.

When a template is inserted in the Java editor and the template pattern contained a template variable, the editor enters the template edit mode.

A box is drawn around all variables. The first variable is selected and can be modified by typing in the editor. If the same variable existed multiple times in the template pattern, all instances of the same variable are highlighted in blue and updated instantaneously to save typing.

Pressing **Tab** navigates to the next unique template variable, **Shift-Tab** navigates to the previous unique template variable.

The template edit mode is left by either pressing **Tab** on the last template variable or pressing **Esc** or **Enter**.

Example:

- Create a method `void m(int[] intarray){}` and position the caret inside the method.
- Type `for` and press **Ctrl+Space** to open Code Assist
- Select the first entry from the list (i.e. *for – iterate over array*). Note the template preview window.
- Note also that the name of the array (i.e. `intarray`) is automatically detected.
- The local variable `i` is now selected and you are in the template edit mode. Typing another name instantaneously updates all occurrences of it.
- Press **Tab**. You can now modify the suggested name for the array (pressing **Shift-Tab** will let you modify the name of the local variable again).
- To leave the template edit mode
 - ◆ press **Tab** or **Enter**, which will move the caret so that you can enter the body of the newly created loop or
 - ◆ press **Esc**, which will not move the caret and preserves the current selection in the editor.

■ Related concepts

[Java editor](#)
[Templates](#)

■ Related tasks

[Using the Java editor](#)
[Writing your own templates](#)

■ Related reference

[Templates preference page](#)

Writing your own templates

You can define your own templates.

- Go to Window > Preferences > Java > Editor > Templates and press the New button.
- In the Name field, enter the name for the template. This name need not be unique. It is used for selecting templates from the Code Assist list.
- Specify the context for the template using the Context combo-box:
 - ◆ Select Java if the template is to be used in normal Java code
 - ◆ Select javadoc if the template is to be used in Javadoc comments
- Uncheck the Automatically insert checkbox, if the template should not be inserted automatically if it is the only proposal available at the caret position.
- In the Description field, enter a brief description of the template.
- Use the Pattern text field to enter the template pattern

The pattern may contain pre-defined and custom template variables.

Template variables are place-holders for the dynamic part of the template pattern, i.e. they are different in every application of the particular template. Before a template is inserted, the variables in its pattern are evaluated and the place-holders are replaced with the evaluated values. Variables are of the form `${variable_name}`.

- ◆ To insert a pre-defined template variable, use the Insert Variable button or press **Ctrl+Space** and select the variable from the presented list.
- ◆ You can insert your own template variables, which will evaluate to the name of the variable itself. You must, however, make sure that the name does not conflict with the pre-defined template variable names in the specific context.
- ◆ If the dollar symbol \$ should be displayed, it must be escaped by using two dollar symbols or by using the variable `${dollar}`.

■ Related concepts

[Templates](#)

■ Related tasks

[Using the Java editor](#)

[Using templates](#)

■ Related reference

[Template preference page](#)

Task Tags

On this preference page, the task tags can be configured. When the tag list is not empty, the compiler will issue a task marker whenever it encounters one of the corresponding tag inside any comment in Java source code. Generated task messages will include the tag and range until the next line separator or comment ending.

See the [Compiler preference page](#) for information on how to enable task tags in your source code.

Action	Description
New...	Adds a new task tag. In the resulting dialog, specify a name and priority for the new task tag.
Remove	Removes the selected task tag.
Edit...	Allows you to edit the selected task tag. In the resulting dialog, edit the name and/or priority for the task tag.
Default	Sets the currently selected task tag as the default task tag. The default task tag is the one that is used in the code templates as specified on the Code Templates preference page . The default task tag is displayed in bold font.

Case sensitivity of the task tags can be specified at the bottom of the preference page using the option ***Case sensitive task tag names***.

■ Related reference

[Java compiler preferences](#)

[Code template preferences](#)

Code templates

This page lets you configure the format of newly generated code and comments.

Code and Comments

The code and comment page contains code templates that are used by actions that generate code. Templates contain variables that are substituted when the template is applied. Some variables are available in all templates, some are specific to templates.

Action	Description
Edit...	Opens a dialog to edit the currently selected code template.
Import...	Imports code templates from the file system.
Export...	Exports all selected code templates to the file system.
Export All...	Exports all code templates to the file system.

Comment templates

Comment templates can contain the variable **`${tags}`** that will be substituted by the standard Javadoc tags (`@param`, `@return`..) for the commented element. The 'Overriding method' comment can additionally contain the template **`${see_to_overridden}`**

- Getter comment: Template that specifies the comment for a getter method
- Setter comment: Template that specifies the comment for a setter method
- Constructor comment: Template that specifies the comment for new constructors
- File comment: Template that specifies the header comment for newly created files. Note that this template can be referenced in the 'New Java File' template
- Type comment: Template that specifies the comment for new types. Note that this template can be referenced in the 'New Java File' template
- Field comment: Template that specifies the comment for new fields. Note that this template can be referenced in the 'New Java File' template
- Method comment: Template that specifies the comment for new methods that do not override an method in a base class
- Overriding method comment: Template that specifies the comment for new methods that override an method in a base class. By default the comment is defined as a non-Javadoc comment (Javadoc will replace this comment with the comment of the overridden method). You can change this to a real Javadoc comment if you want

New Java files template

The 'New Java files' template is used by the New Type wizards when a new Java file is created. The template can specify where comments are added. Note that the template can contain the variable **`${typecomment}`** that will be substituted by the evaluation of the type comment template.

Catch block body template

The 'Catch block body' template is used when a catch block body is created. It can use the variables `${exception_type}` and `${exception_var}`.

Method body template

The 'Method body' templates are used when new method with a body is created that still needs some code to complete its functionality. It contains the variable `${body_statement}` that resolves to a return statement or/and a super-call.

Constructor body template

The 'Constructor body' templates are used when new method or constructor with body is created. It contains the variable `${body_statement}` that resolves a super call.

Getter body template

The 'Getter body' templates are used when new getter method is created . It contains the variable `${body_statement}` that resolves to the appropriate return statement.

Setter body template

The 'Setter body' templates are used when new setter method is created . It contains the variable `${body_statement}` that resolves to the appropriate assignment statement.

Code Template dialog

The following fields and buttons appear in the dialog:

Action	Description
Description	A description of the template
Pattern	The template pattern.
Insert Variables...	Displays a list of pre-defined template specific variables.

■ Related tasks

[Generating getters and setters](#)

■ Related reference

[Source actions](#)

[Java editor](#)

[Java editor preferences](#)

Templates preferences

Code style

The Code style preference page allows to configure naming conventions, style rules and comment settings. These preferences are used when new code has to be generated.

Naming Conventions

The list defines the naming conventions for fields (static and non–static), parameters and local variables. For each variable type it is possible to configure a list of prefix or suffix or both.

Naming conventions are used by all actions and 'Quick Fix' proposals that create fields, parameters and local variables, in particular the [Source actions](#).

Action	Description
Edit...	Opens a dialog to edit the list of prefix and suffixes for the currently selected variable type

Code Conventions

The following settings specify how newly generated code should look like. The names of getter methods can be specified as well as the format of field accesses, method comments, annotations and exception variables.

Action	Description	Default
Qualify field accesses with 'this'	If selected, field accesses are always prefixed with 'this', regardless whether the name of the field is unique in the scope of the field access or not.	Off
Use 'is' prefix for getters returning boolean	If selected, the names of getter methods of boolean type are prefixed with 'is' rather than 'get'.	On
Add comments for new methods and types	If selected, newly generated methods and types are automatically generated with comments where appropriate. See the Code templates preference page to specify the format of the generated comments.	Off
Add '@Override' annotation for overriding methods	If selected, methods which override an already implemented method are annotated with an '@Override' annotation. See the Compiler preference page for settings related to annotations.	On
Exception variable name in catch blocks	Specify the name of the exception variable declared in catch blocks.	e

Related reference

[Source actions](#)

[Java editor](#)

[Java editor preferences](#)

[Java compiler preferences](#)

Code templates preferences

Create Getters and Setters

This dialog lets select the getter and setter methods to create.

Use ***Generate Getters and Setters*** from the Source menu or the context menu on a selected field or type, or a text selection in a type to open the dialog. The Generate Getters and Setters dialog shows getters and setters for all fields of the selected type. The methods are grouped by the type's fields.

The names of the getter and setter methods are derived from the field name. If you use a prefix or suffix for fields (e.g. fValue, _value, val_m), you can specify the suffixes and prefixes in the Code Style preference page (Windows > Preferences > Java > Code Style).

When pressing ***OK***, all selected getters and setters are created.

Option	Description
Select getters and setters to create	A tree containing getter and setter methods that can be created. Getters and setters are grouped by field their associated field.
Select All	Select all getter and setter methods
Deselect All	Deselect all getter and setter methods

You can control whether Javadoc comments are added to the created methods with the ***Generate method comments*** option at the bottom of the dialog.

■ Related tasks

Generating getters and setters

Source actions

String externalization

The Java tools help you to develop applications that can be run on international platforms. An important facet of designing a program for use in different countries is the localization, or externalization, of text that is displayed by the program. By externalizing strings, the text can be translated for different countries and languages without rebuilding the Java program.

The JDT provides the following support for internationalization and string externalization:

- A compiler option lets you mark non-externalized strings as compile-time warnings or errors.
 - ◆ See the Window > Preferences > Java > Compiler > Errors/Warnings > Code style > Usage of non-externalized strings preference
- Tools that allow you to find strings that have not been externalized.
- A wizard that will guide you through externalizing the strings.
- Tools that help you to find unused and incorrectly used keys for strings located in property files.

Comments can be used to denote strings that should not be externalized and should not result in compile-time warnings or errors. These comments are of form `// $NON-NLS-n$` where `n` is the 1-based index of the string in a line of code.

Additional information about internationalized applications can be found in the following documents:

- <http://eclipse.org/articles/Article-Internationalization/how2I18n.html>
- <http://java.sun.com/docs/books/tutorial/i18n/intro/index.html>

■ Related tasks

[Finding strings to externalize](#)

[Finding unused and incorrectly used keys in property files](#)

[Using the Externalize Strings wizard](#)

■ Related reference

[Source menu](#)

[Externalize Strings wizard](#)

[Java Compiler preferences](#)

Finding strings to externalize

To find strings to externalize:

- In a Java view (e.g. Package Explorer), select a set of packages, source folders or projects.
- From the menu bar, select Source > Find Strings to Externalize
- A dialog comes up with a list of all compilation units that have some non-externalized strings
- In the dialog, you can double click on a listed compilation unit or press the **Externalize** button to open the Externalize Strings wizard

■ Related concepts

[String Externalization](#)

■ Related tasks

[Externalizing Strings](#)

[Finding unused and incorrectly used keys in property files](#)

[Using the Externalize Strings wizard](#)

■ Related reference

[Externalize Strings wizard](#)

[Source menu](#)

Externalizing Strings

■ Related concepts

[Java editor](#)

[String Externalization](#)

■ Related tasks

[Finding strings to externalize](#)

[Finding unused and incorrectly used keys in property files](#)

[Using the Externalize Strings wizard](#)

■ Related reference

[Externalize Strings wizard](#)

Finding unused and incorrectly used keys in property files

Finding unused and incorrectly used keys in a property file:

- Open the Search dialog by:
 - ◆ pressing **Ctrl+H** or
 - ◆ selecting Search > Search from the menu bar
- See if a tab called NLS Keys is visible. If it is, then select it.
- If it is not visible, press the Customize button and select the NLS Keys checkbox, press OK to close the dialog and switch to the NLS Key tab.
- In the Resource bundle accessor class field, enter the name of the class that you use to retrieve strings from the property file. You can use the Browse button to select the class from a list.
- In the Property file name field, enter the name of the property file. You can use the Browse button to select the file.
- Select the scope of the search by using the controls in the Scope group.
- Press Search

After the search is finished, the Search Result view displays a list of unused keys in the property file and all incorrect references to non-existing keys.

Note: This feature assumes that the resource bundle accessor class used a method called *getString* with a single *String* parameter to retrieve strings from the property file.

■ Related concepts

[String Externalization](#)

■ Related tasks

[Externalizing Strings](#)

[Finding strings to externalize](#)

[Using the Externalize Strings wizard](#)

■ Related reference

[Externalize Strings wizard](#)

[Source menu](#)

Using the Externalize Strings Wizard

To open the Externalize Strings wizard, do one of the following:

- Find strings to externalize (using the *Find Strings To Externalize* function), select an entry in the resulting dialog and press the *Externalize* button or
- Select the compilation unit in which you want to externalize strings and selecting *Source > Externalize Strings* from the menu bar.

Note: Externalizing strings is undoable (with the same restrictions as for refactorings). Use *Refactor > Undo* from the menu bar to undo externalizing strings.

■ Related concepts

[String Externalization](#)

■ Related tasks

[Externalizing Strings](#)

[Undoing a refactoring operation](#)

[Redoing a refactoring operation](#)

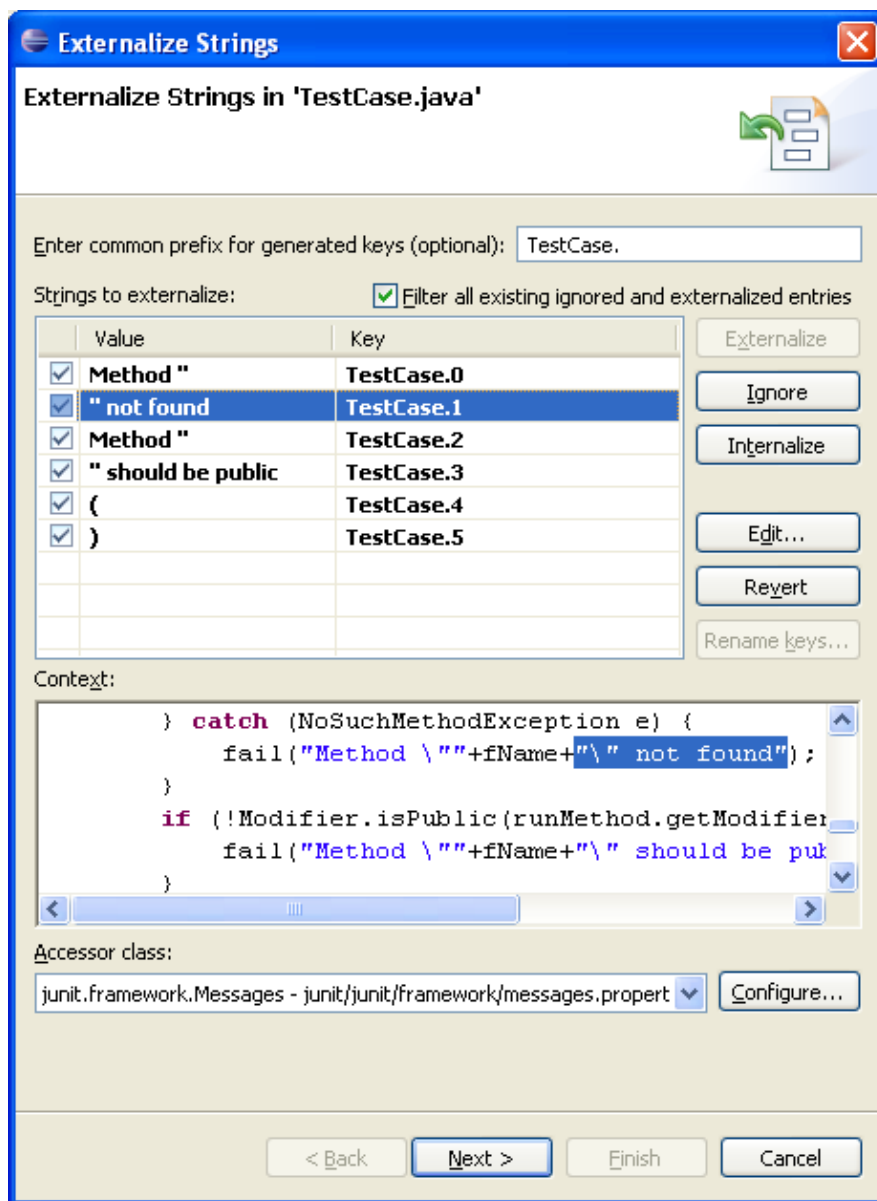
[Key/value page](#)

[Property file page](#)

■ Related reference

[Externalize Strings wizard](#)

Key/value page



Externalize Strings Key/value page

- In the Enter common prefix for generated keys text field, you can specify an optional prefix that will be used for all keys.
- Select one or more entries in the table and:
 - ◆ Press the Translate button to mark them as entries to externalize or
 - ◆ Press the Never Translate button to mark them as entries to be not externalized
 - ◆ Press the Skip button to mark them as entries excluded from externalization
- Icons on the left side of the entries are updated and so are the counter below the table
- To edit a key, single-click on a row in the Key column. You can edit the key in-place. You can also press the **Edit Key** button and edit the key in the dialog that appears then.
- Press Next to proceed to the Property File page or press Finish to externalize strings without checking the settings from the Property File page (if you are not familiar with the externalize strings functionality, it is recommended that you press Next to proceed to the Property File page).

Basic tutorial

Note: You can double-click on the icons that are displayed on the left side of the table entries to alternate the state between Translate, Never Translate and Skip

Note (explanation of the table entry states):

- Strings from entries marked as 'Translate' will be externalized and marked as such in the Java file by adding non-nls tags.
- Strings from entries marked as 'Never Translate' will not be externalized but an non-nls tag will be added to them to inform the wizard that they need not be translated.
- Strings from entries marked as 'Skip' will not be externalized and no tags will be added to them.

■ Related tasks

[Externalizing Strings](#)

[Using the Externalize Strings wizard](#)

[Property file page](#)

[Undoing a refactoring operation](#)

[Redoing a refactoring operation](#)

[Refactoring with preview](#)

[Refactoring without preview](#)

■ Related reference

[Externalize Strings wizard](#)

Property File page

Configure Accessor Class

Resource bundle accessor class (will be created if it does not exist):

Source folder:

Package:

Class name:

Substitution pattern:

Property file location and name:

Source folder:

Package:

Property file name:

Externalize Strings Property File page

1. In the Package field, enter the name for the package in which you want the wizard to locate the property file (you can use the Browse button to display the list of all packages)
2. In the Property file name field, enter the name of the property file (new or existing) in which the externalized strings will be put (you can use the Browse button to display the list of all .properties files located in the package selected in the Package field.)
3. Select the Create resource bundle accessor class checkbox if you want to create a class that will be used to access the externalized strings (Note: the class will be reused if it exists already).
4. In the Class name field, if you have the above-mentioned checkbox selected, you can specify the name of the accessor class
5. Press Next to see a preview of the modifications or press Finish to externalize strings without previewing changes.

Note: The default behavior of the wizard (i.e. creating a class with a name specified in the Class name field and using getString as the name of the method used to retrieve strings from the property file) can be overridden. You may want to do so if you already have an accessor class in another package or your accessor class uses another method with another name to get strings from the property file.

1. Clear the Use default substitution pattern checkbox
2. In the Substitution pattern field enter the new pattern that will be used to retrieve strings from the property file. For each externalized string, the first occurrence of \${key} will be substituted with the key.
3. Use the Add import declaration field if you want the wizard to add an additional import to the compilation unit (you can use the Browse button to help you find a class that you want to import.)

■ Related tasks

[Externalizing Strings](#)

[Using the Externalize Strings wizard](#)

[Key/value page](#)

[Undoing a refactoring operation](#)

[Redoing a refactoring operation](#)

[Refactoring with preview](#)

[Refactoring without preview](#)

■ Related reference

[Externalize Strings wizard](#)

Externalize Strings Wizard

The Externalize Strings wizard allows you to refactor a compilation unit such that strings used in the compilation unit can be translated to different languages. The wizard consists of the following pages:

- String selection page
- Translation settings page
- Error page
- Preview page

String selection page

This page specifies which strings are translated and which not.

Field	Description
Enter common prefix for generated keys	Specifies an optional prefix for every generated key. For example, the fully qualified name of the compilation unit could be used.
Strings to externalize	Displays the list of non-externalized strings with proposed keys and values.
Translate	Marks the selected strings to be translated.
Never Translate	Marks the selected strings as not to be translated.
Skip	Marks the selected strings as to be skipped.
Edit Key...	Opens a dialog for entering a new key.
Context	Displays the occurrence of the string in the context of the compilation unit.

Translation settings page

This page specifies translation specific settings.

Option	Description
Package	Specifies the destination package for the property file.
Property file name	Specifies the property file name.
Create resource bundle accessor class in " <i>package</i> "	If enabled, the wizard creates a class to access the language specific resource bundle.
Class name	The name of the class to access the resource bundle.
Use default substitution pattern	If enabled, the wizard will use default substitution patterns.
Substitution pattern	Specifies the source pattern to replace the string to externalize.
Add import declaration	Specifies additional import declarations. This might be required

	depending on referenced types by the substitution pattern.
--	--

Error page

Displays a list of errors and warnings if any.

Preview page

Displays a preview of the actions which will be performed on 'Finish'.

■ Related concepts

[String externalization](#)

■ Related tasks

[Externalizing Strings](#)

[Using the Externalize Strings wizard](#)

■ Related reference

[Source actions](#)

Viewing marker help

You can use hover help to view various kinds of information in the marker bar in the editor area. For example:

- Information about problems
- Information about breakpoints

Hover your mouse pointer over the marker in the marker bar to view any available hover help.

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Using the Java editor](#)

[Viewing documentation and information](#)

[Viewing Javadoc information](#)

Javadoc location page

This dialog lets you define the location of the Javadoc documentation for a JAR or a Java project.

You can reach this dialog the following ways:

- Select a JAR or Java project, open the context menu and select Properties > Javadoc Location or use **Properties** from the File menu
- In the Javadoc generation wizard, on the Standard doclet settings page, choose **Configure**

Javadoc can be attached to JARs or Java projects. For projects it documents the elements of all source folders, for JARs, elements contained in the JAR are documented. The location is used by

- **Open External Javadoc** in the Navigate menu to find the Javadoc location of an element
- Context Help (F1) to point to a Javadoc location
- Javadoc Export Wizard to link to other documentation or as default destination for a project's documentation

Valid locations are URLs that point to a folder containing the API documentation's *index.html* and *package-list* file. Examples are:

file:///M:/JAVA/JDK1.2/DOCS/API/
http://java.sun.com/j2se/1.4/docs/api/

Option	Description	Default
Javadoc Location	Specify the location of the generated Javadoc documentation. You can Browse in the local file system for a Javadoc location (will result in a file:// URL)	<empty>
Validate	Validate the current location by trying to access the <i>index.html</i> and <i>package-list</i> file with the given URL. If the validation was successful, you can directly open the documentation.	

Javadoc generation

This wizard allows you to generate Javadoc Generation. It is a user interface for the javadoc.exe tool available in the Java JDK. Visit [Sun's Javadoc Tool](#) page for a complete documentation of the Javadoc tool.

First page

Type Selection:

Option	Description
Select types for which Javadoc will be generated	In the list, check or clear the boxes to specify exactly the types that you want to export to the JAR file. This list is initialized by the workbench selection. Only one project can be selected at once as only one project's classpath can be used at a time when running the Javadoc tool.
Create Javadoc for members with visibility	<ul style="list-style-type: none">• Private: All members will be documented• Package: Only members with default, protected or public visibility will be documented• Protected: Only members with protected or public visibility will be documented• Public: Only members with public visibility will be documented (default)
Use Standard Doclet	Start the Javadoc command with the standard doclet (default) <ul style="list-style-type: none">• Destination: select the destination to which the standard doclet will write the generated documentation. The destination is a doclet specific argument, and therefore not enabled when using a custom doclet.
Use Custom Doclet	Use a custom doclet to generate documentation <ul style="list-style-type: none">• Doclet name: Qualified type name of the doclet• Doclet class path: Classpath needed by the doclet class

Standard doclet arguments

Standard Doclet Arguments (available when *Use standard doclet* has been selected):

Option	Description
--------	-------------

Basic tutorial

Document title	Specify a document title.
Generate use page	Selected this option if you want the documentation to contain a Use page.
Generate hierarchy tree	Selected this option if you want the documentation to contain a Tree page.
Generate navigator bar	Selected this option if you want the documentation to contain a navigation bar (header and footer).
Generate index	Selected this option if you want the documentation to contain a Index page.
Generate index per letter	Create an index per letter. Enabled when Generate Index is selected.
@author	Selected this option if you want to the @author tag to be documented.
@version	Selected this option if you want to the @version tag to be documented.
@deprecated	Selected this option if you want to the @deprecated tag to be documented.
deprecated list	Selected this option if you want the documentation to contain a Deprecated page. Enabled when the @deprecated option is selected.
Select referenced classes to which Javadoc should create links	Specify to which other documentation Javadoc should create links when other types are referenced. <ul style="list-style-type: none"> • Select All: Create links to all other documentation locations • Clear All: Do not create links to other documentation locations • Configure: Configure the Javadoc location of a referenced JAR or project.
Style sheet	Select the style sheet to use

General arguments

General Javadoc Options:

Option	Description
Overview	Specifies that Javadoc should retrieve the text for the overview documentation from the given file. It will be placed in overview-summary.html.
Extra Javadoc options	Add any number of extra options here: Custom doclet options, VM options or JRE 1.4 compatibility options.

Basic tutorial

	Note that arguments containing spaces must enclosed in quotes. For arguments specifying html code (e.g. –header) use the or " to avoid spaces and quotes.
Save the settings of this Javadoc export as an Ant script	Specify to store an Ant script that will perform the specified Javadoc export without the need to use the wizard. Existing Ant script can be modified with this wizard (Use Open Javadoc wizard' from the context menu on the generated Ant script)
Open generated index file in browser	Opens the generated index.html file in the browser (Only available when using the standard doclet)

Press ***Finish*** to start the Javadoc generation. If the destination is different to the location configured the project's [Javadoc Location page](#) , you will be asked if you want to set the project's Javadoc location to the new destination folder. By doing this, all invocations of Open External Javadoc will use the now created documentation.

A new process will be started and the generation performed in the background. Open the [Console view](#) (Window > Show View > Console) to see the progress and status of the generation.

■ Related tasks

[Creating Javadoc documentation](#)

■ Related reference

[File actions](#)

[Javadoc Location properties](#)

Creating Javadoc documentation

Select the set (containing one or more elements) of packages, source folders or projects for which you want to generate Javadoc documentation.

Open the Export wizard by doing one of the following:

- Selecting Export from the selection's pop-up menu or
- Selecting File > Export from the menu bar.

In the resulting dialog, select Javadoc from the list and press Next.

■ Related tasks

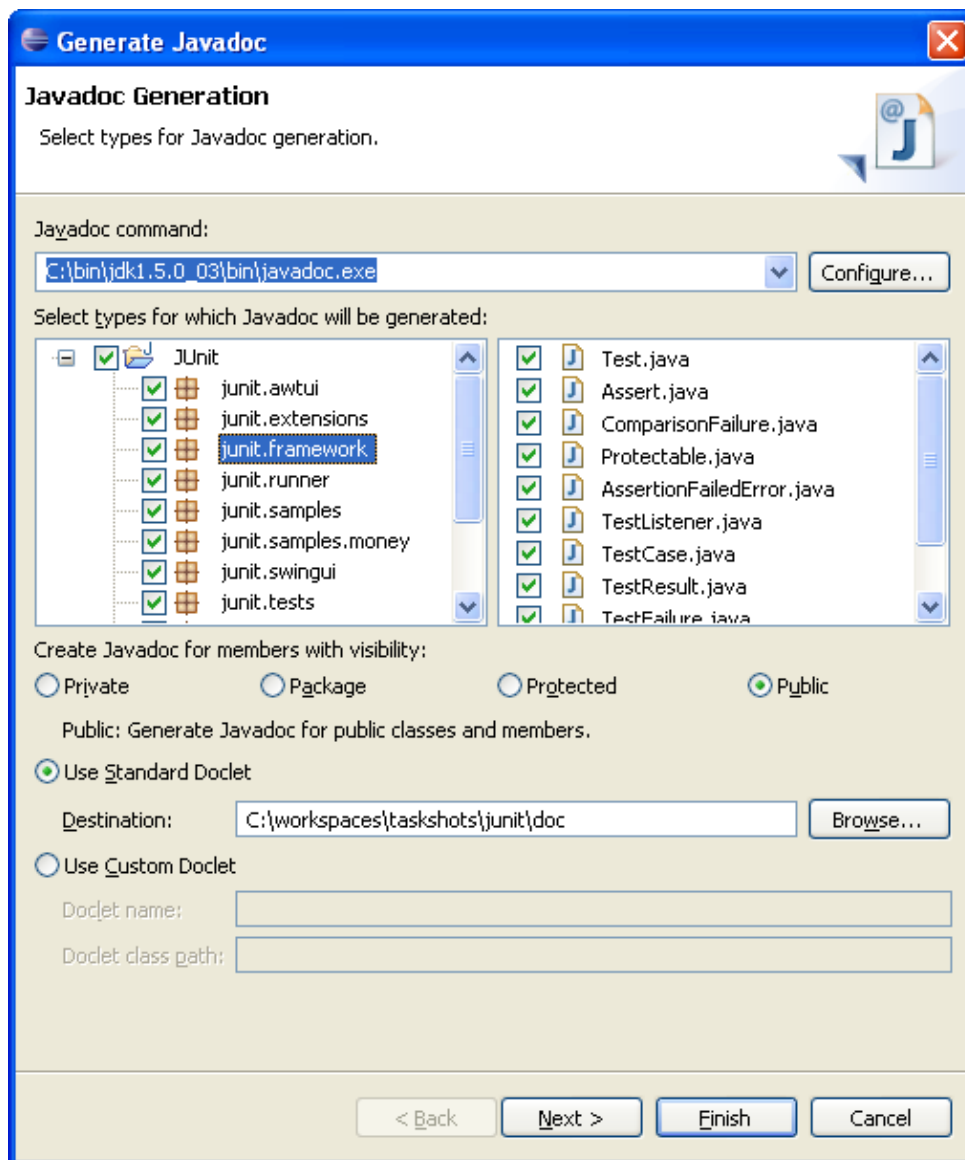
[Selecting types for Javadoc generation](#)

■ Related reference

[Javadoc Generation wizard](#)

[Javadoc Location property page](#)

Selecting types for Javadoc generation



- The JDT uses the Javadoc command (typically available in JRE distributions) to generate Javadoc documentation from source files. To set the location of the Javadoc command, enter the absolute path (e.g. C:\Java\14\jdk1.4\bin\javadoc.exe).
- In the tree control, select the elements for which you want to generate Javadoc.
- Select the visibility using the radio buttons listed under Create Javadoc for members with visibility
- Leave the Use Standard Doclet radio button selected
- Specify the location for the generated Javadoc using the Destination field.
- Press Finish to create generate Javadoc for the elements you selected or press Next to specify more options.

Related tasks

[Creating Javadoc documentation](#)

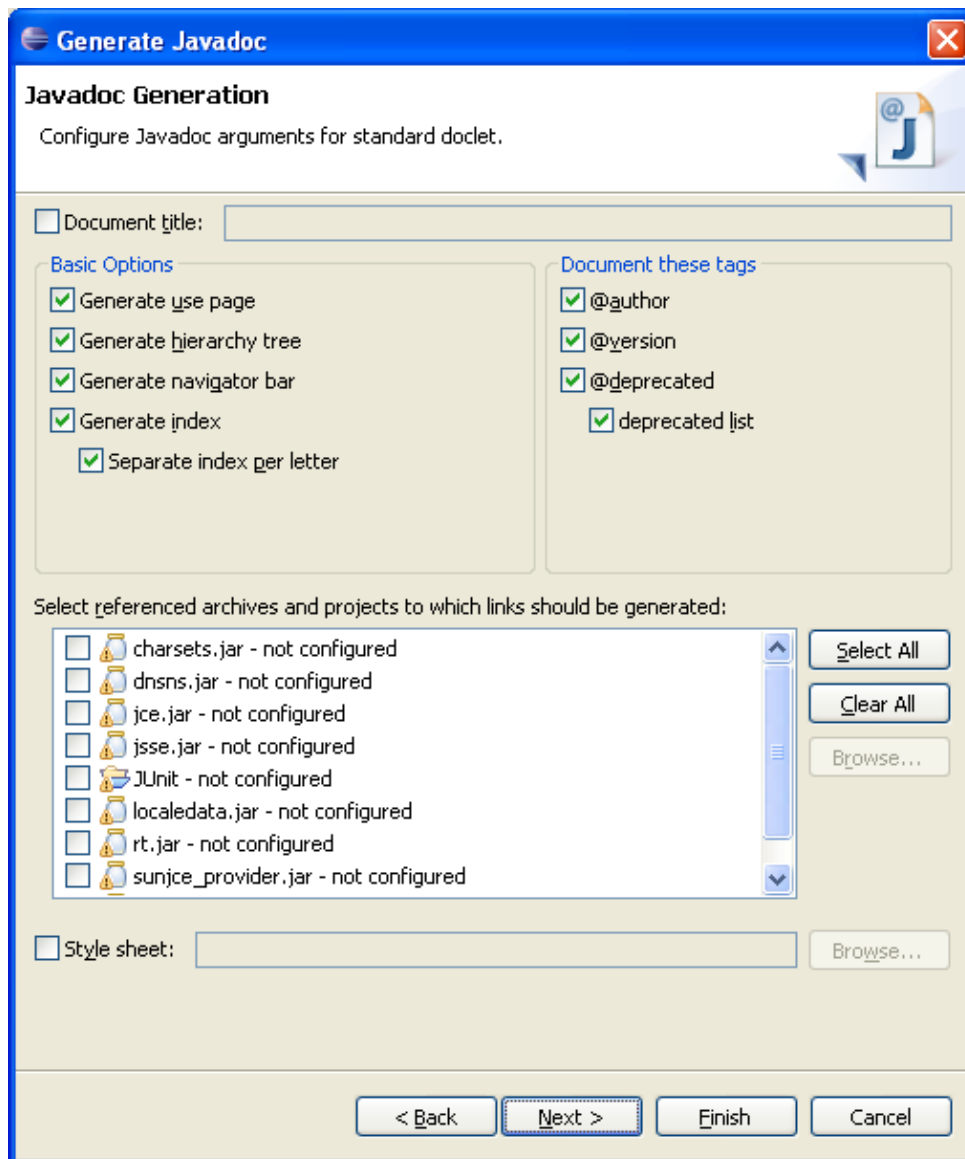
[Configuring Javadoc arguments for standard doclet](#)

■ Related reference

[Javadoc Generation wizard](#)

[Javadoc Location property page](#)

Configuring Javadoc arguments for standard doclet



- Use the checkboxes listed under Basic Options to specify Javadoc options.
- You can change the tags that will be documented by using the checkboxes in the Document these tags group.
- If you want references to classes from a library to be linked to the library's Javadoc, select the library in the list and press Configure to specify the location of the library's Javadoc.
- Press Finish to generate Javadoc or press Next to specify additional Javadoc generation options.

Related tasks

[Creating Javadoc documentation](#)

[Selecting types for Javadoc generation](#)

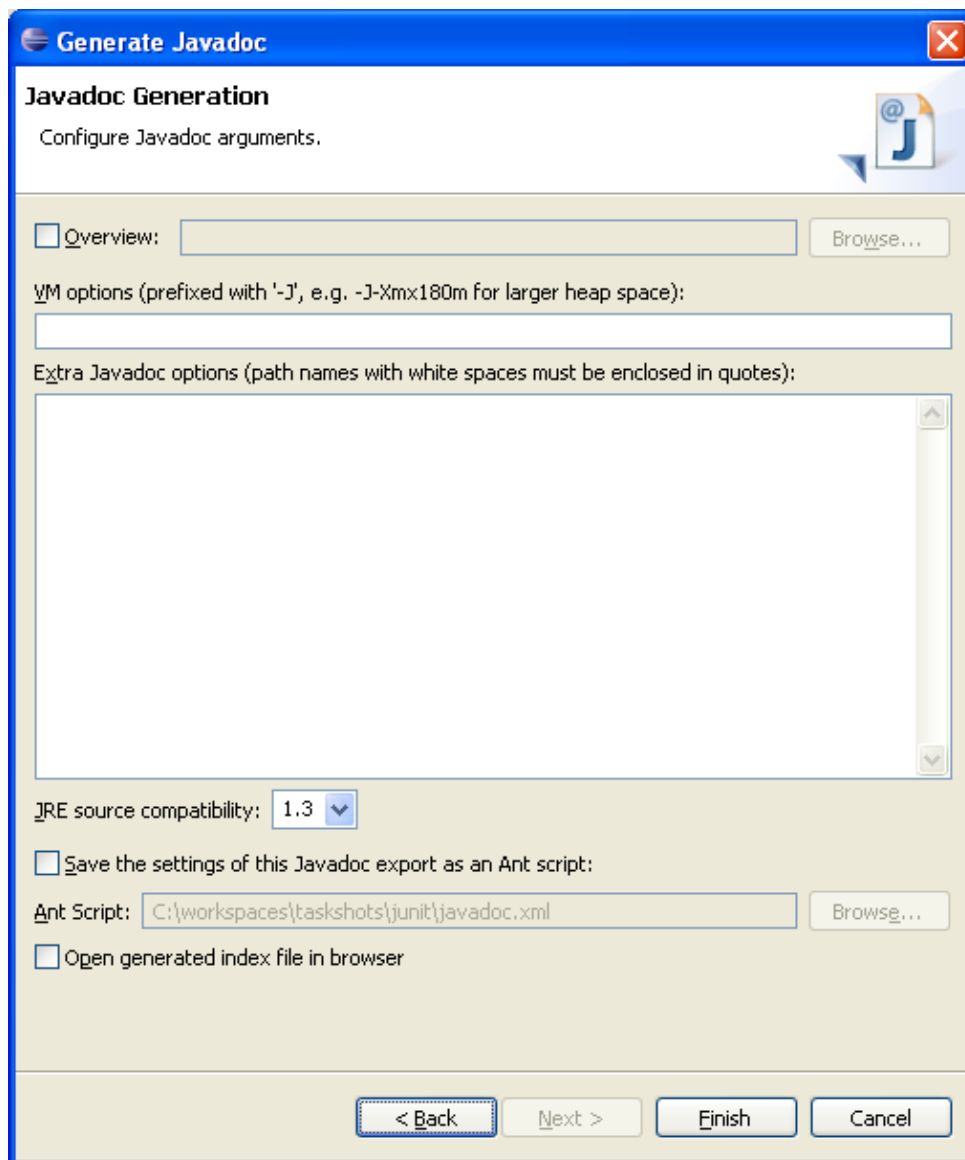
[Configuring Javadoc arguments](#)

■ Related reference

[Javadoc Generation wizard](#)

[Javadoc Location property page](#)

Configuring Javadoc arguments



- You can specify more specific options for the Javadoc command by entering them in the text area.
- Select the JRE source compatibility level.
- Select the Save the settings of this Javadoc export as an Ant script checkbox.
- Specify the location for the Ant Script.
- Select the Open generated index file in browser checkbox.
- Press Finish to generate Javadoc.

Note: The output produced by the Javadoc command (including errors and warning) is shown in the Console view.

■ Related tasks

[Creating Javadoc documentation](#)

[Selecting types for Javadoc generation](#)

Configuring Javadoc arguments for standard doclet

■ Related reference

Javadoc Generation wizard

Javadoc Location property page

Showing and hiding empty packages

To show empty packages:

1. Select the Filters command from the Package Explorer's drop-down menu.
2. In the exclude list clear the checkbox for Empty packages.

To hide empty packages:

1. Select the Filters command from the Package Explorer's drop-down menu.
2. In the exclude list select the checkbox for Empty packages.

■ Related tasks

[Showing and hiding elements](#)

[Filtering elements](#)

■ Related reference

[Java Element Filters](#)

[Package Explorer](#)

Showing and hiding empty parent packages

To show empty parent packages:

1. Select the Filters command from the Package Explorer's drop-down menu.
2. In the exclude list clear the checkbox for Empty parent packages.

To hide empty parent packages:

1. Select the Filters command from the Package Explorer's drop-down menu.
2. In the exclude list select the checkbox for Empty parent packages.

Note: As an example, the parent packages created for package `org.eclipse.ui`, would be:

(default package)

`org`

`org.eclipse`

■ Related tasks

[Showing and hiding elements](#)

[Filtering elements](#)

■ Related reference

[Java Element Filters](#)

[Package Explorer](#)

Showing and hiding Java files

To show Java files:

1. Select the Filters command from the Package Explorer's drop-down menu.
2. In the exclude list clear the checkbox for Java files.

To hide Java files:

1. Select the Filters command from the Package Explorer's drop-down menu.
2. In the exclude list select the checkbox for Java files.

■ Related tasks

[Showing and hiding elements](#)

[Filtering elements](#)

■ Related reference

[Java Element Filters](#)

[Package Explorer](#)

Showing and hiding non-Java elements

To show non-Java elements:

1. Select the Filters command from the Package Explorer's drop-down menu.
2. In the exclude list clear the checkbox for Non-Java elements.

To hide non-Java elements:

1. Select the Filters command from the Package Explorer's drop-down menu.
2. In the exclude list select the checkbox for Non-Java elements.

■ Related tasks

[Showing and hiding elements](#)

[Filtering elements](#)

■ Related reference

[Java Element Filters](#)

[Package Explorer](#)

Showing and hiding non-Java projects

To show non-Java projects:

1. Select the Filters command from the Package Explorer's drop-down menu.
2. In the exclude list clear the checkbox for Non-Java projects.

To hide non-Java elements:

1. Select the Filters command from the Package Explorer's drop-down menu.
2. In the exclude list select the checkbox for Non-Java projects.

■ Related tasks

[Showing and hiding elements](#)

[Filtering elements](#)

■ Related reference

[Java Element Filters](#)

[Package Explorer](#)

Showing and hiding import declarations

To show import declarations:

1. Select the Filters command from the Package Explorer's drop-down menu.
2. In the exclude list clear the checkbox for Import declarations.

To hide non-Java elements:

1. Select the Filters command from the Package Explorer's drop-down menu.
2. In the exclude list select the checkbox for Import declarations.

■ Related tasks

[Showing and hiding elements](#)

[Filtering elements](#)

■ Related reference

[Java Element Filters](#)

[Package Explorer](#)

Showing and hiding package declarations

To show package declarations:

1. Select the Filters command from the Package Explorer's drop-down menu.
2. In the exclude list clear the checkbox for Package declarations.

To hide package declarations:

1. Select the Filters command from the Package Explorer's drop-down menu.
2. In the exclude list select the checkbox for Package declarations.

■ Related tasks

[Showing and hiding elements](#)

[Filtering elements](#)

■ Related reference

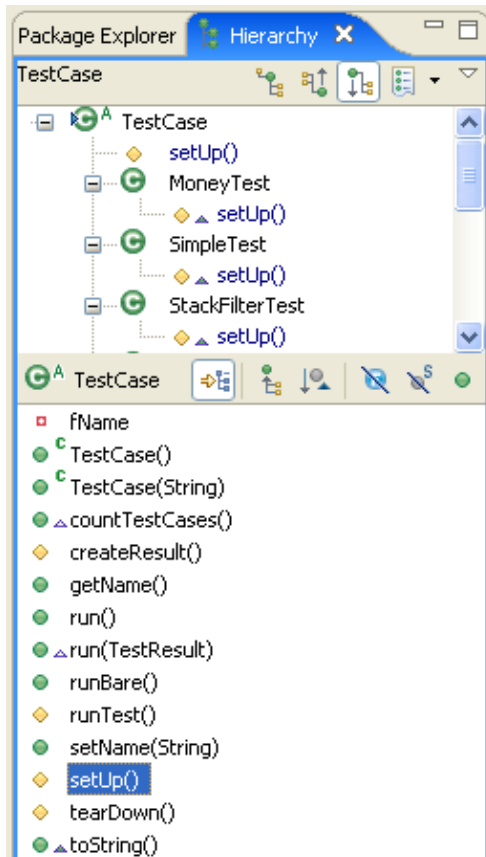
[Java Element Filters](#)

[Package Explorer](#)

Finding overridden methods

You can discover which methods override a selected method.

1. Open the type hierarchy for the selected method's declaring type. Toggle on the *Show the Subtype Hierarchy* toolbar button.
2. In the list pane of the Hierarchy view, make sure that the *Lock View and Show Members in Hierarchy* button is toggled on. This option locks the current class in the method pane and shows only those classes in the upper view that implement the currently selected method of the locked class.



The methods of interest are shown in the upper pane. You can select any method and open it in an editor.

*Note: The selection might not show all members if a filter (such as **Hide Fields** or **Hide Static Members**) is active.*

Related concepts

Java development tools (JDT)

Related tasks

Using the Hierarchy view

Filtering elements

[Opening an editor for a selected element](#)

[Opening a type hierarchy on a Java element](#)

■ Related reference

[Override methods](#)

[Type Hierarchy view](#)

Display view

This view displays the result of evaluating an expression in the context of the current stack frame. You can evaluate and display a selection either from the editor or directly from the Display view.

■ Related concepts

[Java views](#)

[Java perspectives](#)

■ Related tasks

[Evaluating expressions](#)

■ Related reference

[Views and editors](#)

Variables view

This view displays information about the variables in the currently–selected stack frame.

■ Related concepts

[Java views](#)

[Java perspectives](#)

■ Related tasks

[Suspending threads](#)

[Evaluating expressions](#)

■ Related reference

[Views and editors](#)

Show detail pane

This command toggles showing the detail pane for the *Expressions view*. The details pane shows the *toString* for selected objects. For primitive variables, it shows the value.

Code completion in the context of selected variable or the current stack frame is available as well. Evaluation can occur on expressions using *Inspect* and *Display*

■ Related tasks

[Evaluating expressions](#)

Show detail pane

This command toggles showing the detail pane for the *Variables view*. The details pane shows the *toString* for selected objects. For primitive variables, it shows the value.

Code completion in the context of selected variable or the current stack frame is available as well. Evaluation can occur on expressions using *Inspect* and *Display*

■ Related tasks

[Evaluating expressions](#)

Re-launching a program

The workbench keeps a history of each launched and debugged program. To relaunch a program, do one of the following:

- Select a previous launch from **Run** or **Debug** button pull-down menus.
- From the menu bar, select **Run > Run History** or **Run > Debug History** and select a previous launch from these sub-menus.
- In the Debug view, select a process that you want to relaunch, and select **Relaunch** from the process's pop-up menu.

To relaunch the most recent launch, do one of the following:

- Click the **Run** or **Debug** buttons (without using the button pull-down menu).
- Select **Run > Run Last Launched** (**Ctrl+F11**), or **Run > Debug Last Launched** (**F11**) from the workbench menu bar.

■ Related tasks

[Launching a Java program](#)

[Running and debugging](#)

■ Related reference

[Debug view](#)

Console preferences

The following preferences can be set using the Console Preferences page. The console displays output from running applications, and allows keyboard input to be read by running applications.

Option	Description	Default
Fixed width console	This preference controls whether the console has a fixed character width. When on, a maximum character width must also be specified. Some applications write long lines to the console which require horizontal scrolling to read. This can be avoided by setting the console to use a fixed width, automatically wrapping console output.	Off
Limit console output	This preference limits the number of characters buffered in the console. When on, a maximum buffer size must also be specified. When console output surpasses the specified maximum, output is truncated from the beginning of the buffer.	On
Standard Out Text Color	This preference controls the color of text written to the standard output stream by an application.	Blue
Standard Error Text Color	This preference controls the color of text written to the standard error stream by an application.	Red
Standard In Text Color	This preference controls the color of text typed into the console to be read by an application.	Green
Show when program writes to standard out	Often, the Console view is hidden or inactive when you begin running or debugging a program that creates output. If this option is turned on, then when you run a program that produces output, the Console view is automatically opened (if necessary) and is made the active view.	On
Show when program writes to standard error	Often, the Console view is hidden or inactive when you begin running or debugging a program that creates error output. If this option is turned on, then when you run a program that produces error output, the Console view is automatically opened (if necessary) and is made the active view.	On

You can also click the **Change** button to set the font for the Console.

JRE installations

Classpath Variable Preferences

Option	Description
Add...	<p>Adds a new JRE definition to the workbench. In the resulting dialog, specify the following:</p> <ul style="list-style-type: none">• JRE type: (select a VM type from the drop-down list)• JRE name: Type a name for this JRE definition• JRE home directory: Type or browse to select the root directory for this JRE installation• Javadoc URL: Type or browse to select the URL location. The location is used by the Javadoc export wizard as a default value and by the 'Open External Javadoc' action.• Debugger timeout: Type the default timeout for this JRE's debugger (in ms)• Either check the box to use the default library locations for this JRE or clear the checkbox and type or browse to select library locations for the following:<ul style="list-style-type: none">◆ JAR file (e.g., classes.zip)◆ Source file (e.g., source.zip) <p>You can click the Browse buttons to browse for paths.</p>
Edit...	Allows you to edit the selected JRE.
Remove	Removes the selected JRE from the workbench.
Search...	Automatically searches for JREs installed in the local file system and creates corresponding JRE definitions in the workspace.

■ Related concepts

Classpath variables

■ Related tasks

Working with JREs

■ Related reference

Source Attachment

Source attachments

To browse the source of a type contained in library you can attach a source archive or source folder to this library. The editor will then show the source instead of a the decompiled code. Having the source attachment set the debugger can offer source level stepping in this type.

The Source Attachment dialog can be reached in several ways:

- Select a JAR in the Package Explorer and choose **Properties > Java Source Attachment** from the context menu or the [Project menu](#)
- Open the Java Build Path page of a project (**Projects > Properties > Java Build Path**). On the **Libraries** page expand the library's node and select the **Source attachment** attribute and press **Edit**
- Open an editor on a class file. If the source attachment has not already been configured for this JAR, the editor contains a button Attach Source

Depending of how a JAR was contributed to the classpath, you can see different types of Source attachment dialogs:

JAR

In the Location path field, enter the path of an archive or a folder containing the source. Use either the **Workspace**, **External File** or the **External Folder** button to browse for a location.

Variable

In the Location Variable Path field enter a *variable path* that points to the source attachment's location. A variable path has as first segment a variable (which will resolve to a folder or file), the rest is an optional path extension (e.g.*MYVARIABLE/src.jar*). Use either the Variable button to select an existing variable and the Extension button to select the extension path. The Extension button is only enabled when the variable can be extended (resolves to a folder)

JRE_SRC is a reserved variable that points to a JRE selected in the [Installed JREs preference page](#) (Window > Preferences > Java > Installed JREs). Go to this preference page to configure the source attachment for the JRE's library..

■ Related concepts

[Build classpath](#)

■ Related tasks

[Working with build paths](#)

[Attaching source to variables](#)

[Attaching source to a JAR file](#)

■ Related reference

Installed JREs preferences

Java Build Path properties

Editing a JRE definition

You can modify all settings for a JRE definition except its JRE type.

1. From the menu bar, select **Window > Preferences**.
2. In the left pane, expand the **Java** category, and select **Installed JREs**.
3. Select the JRE definition that you want to edit and click **Edit...** The Edit JRE page opens.
4. In the **JRE name** field, edit the name for the JRE definition. All JREs of the same type must have a unique name.
5. In the **JRE home directory** field, edit or click **Browse...** to select the path to the root directory of the JRE installation (usually the directory containing the *bin* and *lib* directories for the JRE). This location is checked automatically to make sure it is a valid path.
6. In the **Javadoc URL** field, edit or click **Browse...** to select the URL location. The location is used by the Javadoc export wizard as a default value and by the 'Open External Javadoc' action.
7. If you want to use the default libraries and source files for this JRE, select the **Use default system libraries** checkbox. Otherwise, clear it and customize as desired. Source can be attached for the referenced jars as well.
8. Click **OK** when you are done.

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Adding a new JRE definition](#)

[Deleting a JRE definition](#)

[Overriding the default system libraries for a JRE definition](#)

[Working with JREs](#)

■ Related reference

[Installed JREs preference page](#)

Deleting a JRE definition

You can delete Java runtime environment definitions that are available for executing Java programs in the workbench.

1. Select **Window > Preferences** from the main menu bar.
2. From the left pane, expand the **Java** category and select **Installed JREs**.
3. Select the definition you want to delete and click **Remove**.
4. Check the box for the definition that you want to use as the default JRE for the workbench.

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Working with JREs](#)

[Adding a new JRE definition](#)

[Editing a JRE definition](#)

■ Related reference

[Installed JREs preference page](#)

Overriding the default system libraries for a JRE definition

You can override the system libraries and the corresponding source attachments when adding a JRE.

1. Begin adding a new JRE definition.
2. In the Create JRE dialog, clear the *Use default system libraries* checkbox.
If the system was able to determine the set of system libraries, these libraries will already be displayed.
3. Order and/or remove the system libraries that were determined. Add external JARs to the list of system libraries.
4. Associate source with the system libraries using the *Attach source* button.
5. When you are finished with this dialog, click **OK** to create the new JRE definition. The new JRE definition will use the customized system libraries (and attached source).

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Working with JREs](#)

[Adding a new JRE definition](#)

[Editing a JRE definition](#)

■ Related reference

[Installed JREs preference page](#)

Installed JREs

Check the box for the JRE that you want to act as your default for running and debugging Java programs in the workbench.

This JRE defines the values for the three reserved classpath variables (JRE_LIB, JRE_SRC, JRE_SRCROOT).

Defining the JAR file's manifest

You can either define the important parts of the JAR file manifest directly in the wizard or choose to use a manifest file that already exists in your workbench.

Creating a new manifest

1. Follow the procedure for creating a JAR file, but click *Next* in the last step to go to the JAR Packaging Options page.
2. Set any advanced options that you want to set, and then click *Next* again to go to the JAR Manifest Specification page.
3. If it is not already selected, click the *Generate the manifest file* button.
4. You can now choose to save the manifest in the workbench. This will save the manifest for later use. Click *Save the manifest in the workspace*, then click *Browse* next to the *Manifest file* field to specify a path and file name for the manifest.
5. If you decided to save the manifest file in the previous step and you chose to save the JAR description on the previous wizard page, then you can choose to reuse it in the JAR description (by selecting the *Reuse and save the manifest in the workspace* checkbox). This means that the saved file will be used when the JAR file is recreated from the JAR description. This option is useful if you want to modify or replace the manifest file before recreating the JAR file from the description.
6. You can choose to seal the JAR and optionally exclude some packages from being sealed or specify a list with sealed packages. By default, nothing is sealed.
7. Click the *Browse* button next to the *Main class* field to specify the entry point for your applications.
Note: If your class is not in the list, then you forgot to select it at the beginning.
8. Click *Finish*. This will create the JAR, and optionally a JAR description and a manifest file.

Using an existing manifest

You can use an existing manifest file that already exists in your workbench.

1. Follow the procedure for creating a JAR file, but click *Next* in the last step to go to the JAR Packaging Options page.
2. Set any advanced options that you want to set, and then click *Next* again to go to the JAR Manifest Specification page.
3. Click the *Use existing manifest from workspace* radio button.
4. Click the *Browse* button to choose a manifest file from the workbench.
5. Click *Finish*. This will create the JAR and optionally a JAR description.

Related concepts

[Java development tools \(JDT\)](#)

Related tasks

[Creating a new JAR file](#)

[Setting advanced options](#)

■ Related reference

JAR file exporter

Setting advanced options

1. Follow the procedure for creating a JAR file, but click *Next* in the last step to go to the JAR Packaging Options page.
2. If you want to save the JAR file description, select the *Save the description of this JAR in the workspace* checkbox.
3. The compiler is able to generate CLASS files even when source contains errors. You have the option to exclude CLASS (but not source) files with compile errors. These files will be reported at the end, if reporting is enabled.
4. You can choose to exclude CLASS (but not source) files that have compile warnings. These files will be reported at the end.
Note: This option does not automatically exclude class files with compile errors.
5. You can choose to include the source folder path by selecting the *Create source folder structure* checkbox.
6. Select the *Build projects if not built automatically* checkbox if you want the export to perform a build before creating the JAR file.
7. Click *Finish* to create the JAR file immediately or *Next* if you want to change the default manifest.

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Creating a new JAR file](#)

[Defining the JAR file's manifest](#)

■ Related reference

[JAR file exporter](#)

JAR file exporter

This wizard allows you to create a JAR file.

JAR package specification

JAR Specification Options:

Option	Description
Select the resources to export	In the list, check or clear the boxes to specify exactly the files that you want to export to the JAR file. This list is initialized by the workbench selection.
Export generated class files and resources	If you want to export generated CLASS files and resources, select this option.
Export Java source files and resources	If you want to export JAVA source files and resources, select this option.
Select the export destination	Enter an external file system path and name for a JAR file (either new or existing). Either type a valid file path in the field or click Browse to select a file via a dialog.
Options	<p>You can select any of the following options:</p> <ul style="list-style-type: none">• Compress the contents of the JAR file : to create a compressed JAR file• Overwrite existing files without warning : if an existing file might be overwritten, you are prompted for confirmation. This option is applied to the JAR file, the JAR description, and the manifest file.

JAR packaging options

JAR Options:

Option	Description
Select options for handling problems	<p>Choose whether to export classes with certain problems:</p> <ul style="list-style-type: none">• Export class files with compile errors• Export class files with compile warnings
Create source folder structure	Selected this option if you want the source folder structure to be rebuilt in the JAR. This option is only enabled when source files but no class files are exported.
Build projects if not build automatically	Select this option to force a rebuild before exporting. It is recommended to build before exporting so the exported class files are up to date.
Save the description of this JAR in the workspace	If you select this option, you can create a file in the workbench describing the JAR file you are creating. Either type and/or browse to select a path and name for this new file.

JAR manifest specification

JAR Manifest Specification Options (Available when class file are exported):

Option	Description
Specify the manifest	<p>Choose the source of the manifest file for this JAR file:</p> <ul style="list-style-type: none"> • Generate the manifest file (you can also choose either to save or reuse and save the new manifest file) • Use existing manifest from workspace
Seal contents	<p>Choose which packages in the JAR file must be sealed:</p> <ul style="list-style-type: none"> • Seal the JAR: to seal the entire JAR file; click <i>Details</i> to exclude selectively • Seal some packages; click <i>Details</i> to choose selectively <p><i>Note: This option is only available if the manifest is generated.</i></p>
Select the class of the application entry point	<p>Type or click <i>Browse</i> to select the main class for the JAR file, if desired.</p> <p><i>Note: This option is only available if the manifest is generated.</i></p>

■ Related tasks

Creating JAR Files

■ Related reference

File actions

Creating JAR files

You can create and regenerate JAR files in the workbench.

■ Related tasks

[Creating a new JAR file](#)

[Regenerating a JAR file](#)

■ Related reference

[JAR file exporter](#)

Regenerating a JAR file

You can use a JAR file description to regenerate a previously created JAR file.

1. Select one or more JAR file descriptions in your workbench.
2. From the selection's pop-up menu, select **Create JAR**. The JAR file(s) are regenerated.

■ Related concepts

Java development tools (JDT)

■ Related tasks

Creating a new JAR file

Defining the JAR file's manifest

■ Related reference

JAR file exporter

Adding source code as individual files

From a ZIP or JAR file

1. In the Navigator or Package Explorer, you can optionally select the source folder into which you wish to import the source. *Note: It is important to choose a source container; otherwise, the imported files will not be on the build path.*
2. From the workbench menu bar, select **File > Import**.
3. In the Import wizard, select **Zip file**, then click **Next**.
4. In the **Zip file** field, type or browse to select the file from which you would like to add the resources. In the import selection panes, use the following methods to select exactly the resources you want to add:
 - ◆ Expand the hierarchies in the left pane and check or clear the boxes representing the containers in the selected directory. Then in the right pane, select or clear boxes for individual files.
 - ◆ Click **Select Types** to select specific file types to add.
 - ◆ Click **Select All** to select all files in the directory, then go through and deselect the ones that you don't want to add.
 - ◆ Click **Deselect All** to deselect all files in the directory, then go through and choose individual files to add.
5. In the **Select the destination for imported resources** field, type or browse to select a workbench container for the added resources (if it was not already selected when you activated the **Import** command).
6. In the **Options** area, you can choose whether or not to overwrite existing resources without warning.
7. Click **Finish** when you are done.

From a directory

1. In the Navigator or Package Explorer, select the source folder into which you wish to import the source. *Note: It is important to choose a source container; otherwise, the imported files will not be on the build path.*
2. From the workbench menu bar, select **File > Import**.
3. In the import wizard, select **File System**, then click **Next**.
4. In the **Directory** field, type or browse to select the directories from which you would like to add the resources.
5. In the import selection panes, use the following methods to select exactly the resources you want to add:
 - ◆ Expand the hierarchies in the left pane and check or clear the boxes representing the containers in the selected directory. Then in the right pane, select or clear boxes for individual files.
 - ◆ Click **Select Types** to select specific file types to add.
 - ◆ Click **Select All** to select all files in the directory, then go through and deselect the ones that you don't want to add.
 - ◆ Click **Deselect All** to deselect all files in the directory, then go through and choose individual files to add.
6. In the **Select the destination for imported resources** field, type or browse to select a workbench container for the added resources (if it was not already selected when you activated the **Import** command).
7. In the **Options** area, you can choose whether you want to:

Basic tutorial

- ◆ overwrite existing resources without warning
 - ◆ create a complete file structure for the imported files
8. Click ***Finish*** when you are done.

■ Related concepts

[Java development tools \(JDT\)](#)
[Build classpath](#)

■ Related tasks

[Adding a JAR file as a library](#)
[Working with build paths](#)

■ Related reference

[Java Build Path](#)
[Package Explorer](#)

Adding a JAR file as a library

To add a JAR file as a library, you can either drag and drop the JAR file into the workbench from the file system or you can use the Import wizard to import the file.

1. From the workbench menu bar, select **File > Import**. The Import wizard opens
2. Select **File System**, then click **Next**.
3. In the **From directory** field, type or browse to select the directory where the JAR file resides.
4. In the import selection panes, expand the hierarchy and use the buttons to select the JAR file you want to import.
5. In the **Into folder** field, type or browse to select a workbench container for the JAR file.
6. Click **Finish** when you are done.
7. You now must add the JAR file to the build class path of any projects that need this library.

■ Related concepts

[Java builder](#)

[Build classpath](#)

■ Related tasks

[Adding a JAR file to the build path](#)

[Attaching source to a JAR file](#)

[Working with build paths](#)

[Adding source code as individual files](#)

■ Related reference

[Java Build Path](#)

Java Compiler page

The options in this page indicate the compiler settings for a Java project. You can reach this page through the

- Java Compiler property page (File > Properties > Java Compiler) from the context menu on a created project or the [File menu](#)

A project can either reuse workspace default settings or use its own custom settings.

Option	Description
Enable project specific settings	Once selected, compiler settings can be configured for this project. All Java compiler preferences can be customized. At any time, it is possible to revert to workspace defaults, by using the button <i>Restore Defaults</i> .

■ Related concepts

[Build classpath](#)

■ Related tasks

[Java compiler preferences](#)

[Java Build Path properties](#)

[Frequently asked questions on JDT](#)

Converting line delimiters

The Java editor can handle Java files with heterogeneous line delimiters. However other external editors or tools may not be able to handle them correctly.

If you want to change line delimiters used by your Java files:

1. Open a Java file in the Java editor
2. Select **File > Convert Line Delimiters To** from the menu bar
3. Select the delimiters to which you want to convert the file to (one of *Windows*: CRLF, *Unix*: LF, *MacOS 9*: CR)

Notes:

- You can also apply these conversions to multiple files, whole folders or project that are selected in the Package Explorer or in the Navigator.
- To change the line delimiters used for new files,
 - ◆ set the workspace default (**Window > Preferences > General > Editors > New text file line delimiter**), or
 - ◆ set the project-specific default (**Context menu > Properties > Info > New text file line delimiter**)

■ Related concepts

Java editor

■ Related tasks

Using the Java editor

■ Related reference

Source menu

Finding and replacing

■ Related tasks

[Using the Java editor](#)

[Using the Find/Replace dialog](#)

[Using Incremental Find](#)

[Finding Next or Previous Match](#)

Using the Find/Replace dialog

To open the Find / Replace dialog:

1. Optionally, select some text in a text (or Java) editor
2. Press **Ctrl+F** or
3. From the menu bar, select Edit > Find / Replace

To use the Find / Replace dialog:

1. In the Find field, enter the text you want to replace
2. Optionally, in the Replace With field, enter the next text
3. Use the radio buttons in the Direction group to select the direction of finding occurrences (relative to the current caret position)
4. In the Scope radio button group:
 - ◆ Select the Selected Lines button if you want to limit find / replace to the lines you selected when opening the dialog
 - ◆ Select the All button otherwise
5. In the Options group:
 - ◆ Select the Case Sensitive checkbox if you want finding to be case sensitive.
 - ◆ Select the Whole Word checkbox if you want to find / replace only those occurrences in which the text you entered in the Find field is a whole word
 - ◆ Select the Wrap Search checkbox if you want the dialog to continue from the top of the file once the bottom is reached (if you find / replace in Selected Lines only, then this option applies to those lines only)
 - ◆ Select the Incremental checkbox if you want to perform incremental find (e.g. the dialog will find matching occurrences while you are typing in the Find field)
 - ◆ Select the Regular Expressions checkbox if you want to perform regex search. When Regular Expressions are enabled, you can use content assist (**Ctrl+Space**) in the Find and Replace fields to get support for regex syntax.
6. Press:
 - ◆ Find if you want to find the next matching occurrence
 - ◆ Replace, if you want to replace the currently selected occurrence with the new text
 - ◆ Replace / Find, if you want to replace the currently selected occurrence with the new text and find the next matching occurrence
 - ◆ Replace All, if you want to replace all matching occurrences with the new text
7. Press Close.

■ Related concepts

[Java editor](#)

■ Related tasks

[Using the Java editor](#)

[Finding and replacing](#)

■ Related reference

[Edit menu](#)

Using Incremental Find

To use Incremental Find:

1. In the text (or Java) editor, press **Ctrl+J** or select Edit > Incremental Find Next from the menu bar.
2. The workbench status line displays "Incremental Find:". The editor is now in the Incremental Find mode.
3. As you type, the editor finds the next occurrence of the text and updates the selection after each character typed.
4. Navigate to the next or previous match by pressing **Arrow Down** or **Arrow Up**.
5. Undo the last action within the Incremental Find mode by pressing **Backspace**.
6. You can leave the Incremental Find mode by pressing **Esc**.

■ Related concepts

[Java editor](#)

■ Related tasks

[Using the Java editor](#)

[Finding and Replacing](#)

[Finding Next or Previous Match](#)

■ Related reference

[Edit menu](#)

Finding next or previous match

To find the next match:

1. In the text (or Java) editor, press **Ctrl+K** or select Edit > Find Next from the menu bar.
2. The next occurrence of the text selected in the editor will be found.

To find the previous match:

1. In the text (or Java) editor, press **Ctrl+Shift+K** or select Edit > Find Previous from the menu bar.
2. The next occurrence of the text selected in the editor will be found.

■ Related concepts

[Java editor](#)

■ Related tasks

[Using the Java editor](#)

[Finding and Replacing](#)

[Using Incremental Find](#)

■ Related reference

[Edit menu](#)

Changing the encoding used to show the source

To change the encoding used by the Java editor to display source files:

- With the Java editor open and the file saved, select **Edit > Set Encoding...** from the menu bar
- Select **Other** and then select an encoding from the menu or type in the encoding's name.

Notes:

- This setting affects only the way the source is presented—it does not change the file's contents.
- To change the default encoding used for interpreting text files,
 - ◆ set the workspace default (**Window > Preferences > General > Editors > Text file encoding**), or
 - ◆ set the project- or folder-specific default (**Context menu > Properties > Info > Text file encoding**)

This will change the encoding of all contained files that don't have an explicit encoding set.

■ Related concepts

Java editor

■ Related tasks

Using the Java editor

■ Related reference

Edit menu

Commenting and uncommenting lines of code

To toggle line comments (`// comment`) in the Java editor:

- Select the lines you wish to comment or uncomment, or set the caret into a line
- Do one of the following:
 - ◆ Press **Ctrl+/** or
 - ◆ Select **Source > Toggle Comment** from the menu bar

To insert a block comment (`/*comment*/`) in the Java editor:

- Select the text range you wish to comment
- Do one of the following
 - ◆ Press **Ctrl+Shift+/** or
 - ◆ Select **Source > Add Block Comment** from the menu bar

To remove a block comment (`/*comment*/`) in the Java editor:

- Select the text range you wish to uncomment or set the caret into a block comment
- Do one of the following
 - ◆ Press **Ctrl+Shift+** or
 - ◆ Select **Source > Remove Block Comment** from the menu bar

■ Related concepts

[Java editor](#)

■ Related tasks

[Using the Java editor](#)

■ Related reference

[Source menu](#)

Shifting lines of code left and right

To shift lines of code to the right (i.e. indent):

- Select the lines you wish to shift right
- Do one of the following:
 - ◆ Press **Tab** or
 - ◆ Select **Source > Shift Right** from the menu bar

To shift lines of code to the left (i.e. outdent):

- Select the lines you wish to shift left
- Do one of the following:
 - ◆ Press **Shift+Tab** or
 - ◆ Select **Source > Shift Left** from the menu bar

■ Related concepts

[Java editor](#)

■ Related tasks

[Using the Java editor](#)

■ Related reference

[Source menu](#)

Exclusion and inclusion filters

Exclusion and inclusion filters are Ant file patterns that further define the set of source files to be considered by the Java builder and other JDT tools. To allow nesting of source folders inside each other, or simply to exclude parts of a source tree, one can associate exclusion filters to a source folder. Symmetrically inclusion filters can be used to include another part of a source tree. The Java builder will ignore source files that are excluded, and the ones that are not included.

■ Related concepts

[Java builder](#)

[Access rules](#)

■ Related tasks

[Building a Java program](#)

[Viewing and editing a project's build path](#)

[Working with build paths](#)

[Creating a new source folder with exclusion filter](#)

■ Related reference

[Java Build Path properties](#)

Access rules

Access rules are Ant patterns defined on build classpath entries that tells the compiler to signal the use of types matching the patterns.

- Non-accessible rules define types that must not be referenced.
- Discouraged rules define types that should not be referenced.
- Accessible rules define types that can be referenced.

■ Related concepts

[Java builder](#)

[Exclusion and inclusion filters](#)

■ Related tasks

[Adding a JAR file to the build path](#)

[Adding a library folder to the build path](#)

[Building a Java program](#)

[Viewing and editing a project's build path](#)

[Working with build paths](#)

■ Related reference

[Java Build Path properties](#)

Creating a new source folder with exclusion filter

In a project that uses source folders, you can create a new folder to contain Java source code with exclusion patterns. Exclusion patterns are useful if you have nested source folders. There are two ways to do it:

1. You don't already have an existing Java project in which you want to create a source folder with exclusion pattern.
2. You already have an existing Java project.

Starting from scratch

1. First follow the steps of the task "Creating a new source folder".
2. Once a first source folder is created, if you want to create another source folder that is nested inside the first one, you need to use the exclusion pattern.
3. Open the [Java Build Path](#) page (**Project > Properties > Java Build Path**) and click the **Source** tab.
4. Click **Add Folder**, select an existing source folder, and click **Create New Folder** to create a folder that is nested inside the first one.
5. You will get a dialog saying that exclusion filter has been added to the first source folder.

NOTE: The trailing '/' at the end of the exclusion pattern **is required** to exclude the children of this folder. The exclusion pattern follows the ant exclusion pattern syntax.

6. Click **OK** and **Finish** when you are done.

From an existing Java Project

1. Before you start, make sure that your project properties are enabled to handle exclusion filters in source folders.
2. In the Package Explorer, select the project where you want the new source folder to reside.
3. From the project's pop-up menu, select **New > Source Folder**. The New Source Folder wizard opens.
4. In the **Project Name** field, the name of the selected project appears. If you need to edit this field, you can either type a path or click **Browse** to choose a project that uses source folders.
5. In the **Folder Name** field, type a name for the new source folder. If you choose a path that is nested inside an existing source folder, you will see an error saying that you have nested source folders.
6. Check **Update exclusion filters in other source folders to solve nesting**.
7. Click **Finish** when you are done.

■ Related concepts

[Java projects](#)

■ Related tasks

[Creating Java elements](#)

[Creating a new Java project](#)

[Creating a new Java package](#)

[Creating a Java project as its own source container](#)

[Creating a Java project with source folders](#)

[Creating a new source folder](#)

■ Related reference

[New Source Folder wizard](#)

[Java Toolbar actions](#)

[Package Explorer](#)

Creating a new source folder with specific output folder

Any source folder can use either the default project output folder or its specific output folder.

1. First follow the steps of the task "Creating a new source folder".
2. Once a first source folder is created, you might want to change its default output folder.
3. Click on ***Allow output folders for source folders***. This adds a new entry to the source folder tree.
4. Expand the source folder tree.
5. Double-click on the ***Output folder entry***.
6. A dialog asks you if you want to use the project's default output folder or a specific output folder. Choose the second option and click ***Browse....***
7. Select the folder you want and click ***OK*** and then ***Finish***.

■ Related concepts

[Java projects](#)

■ Related tasks

[Creating Java elements](#)

[Creating a new Java project](#)

[Creating a new Java package](#)

[Creating a Java project as its own source container](#)

[Creating a Java project with source folders](#)

[Creating a new source folder](#)

■ Related reference

[Java Build Path](#)

[New Source Folder wizard](#)

[Java Toolbar actions](#)

[Package Explorer](#)

Creating your first Java project

In this section, you will create a new Java project. You will be using JUnit as your example project. JUnit is an open source unit testing framework for Java.

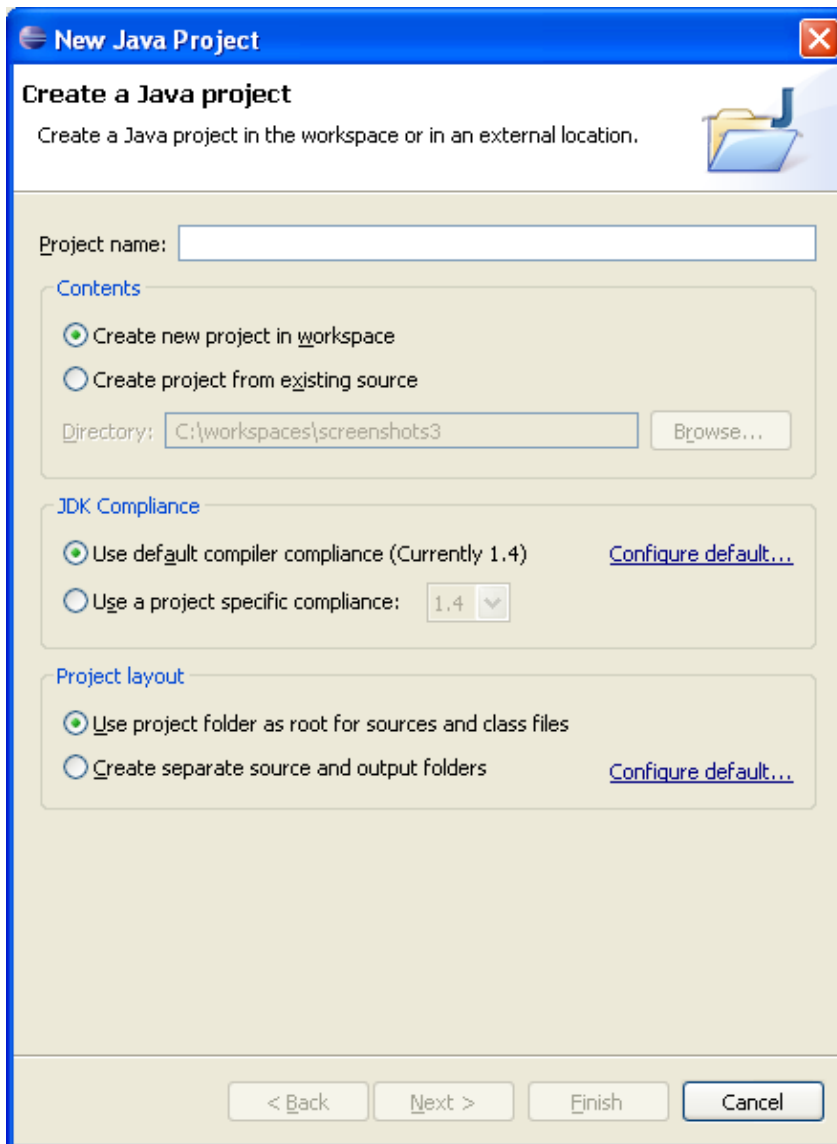
Getting the Sample Code (JUnit)

First you need to download the JUnit source code.

1. Go to the <http://www.eclipse.org/downloads/> page
2. Select the Downloads link under Java Development Tools.
3. Select the link for the release that you are working with.
4. Scroll down to the *Example Plug-ins* section and download the examples archive.
5. Extract the contents of the Zip file to a directory from now on referenced as <Downloads> (e.g. c:\myDownloads).

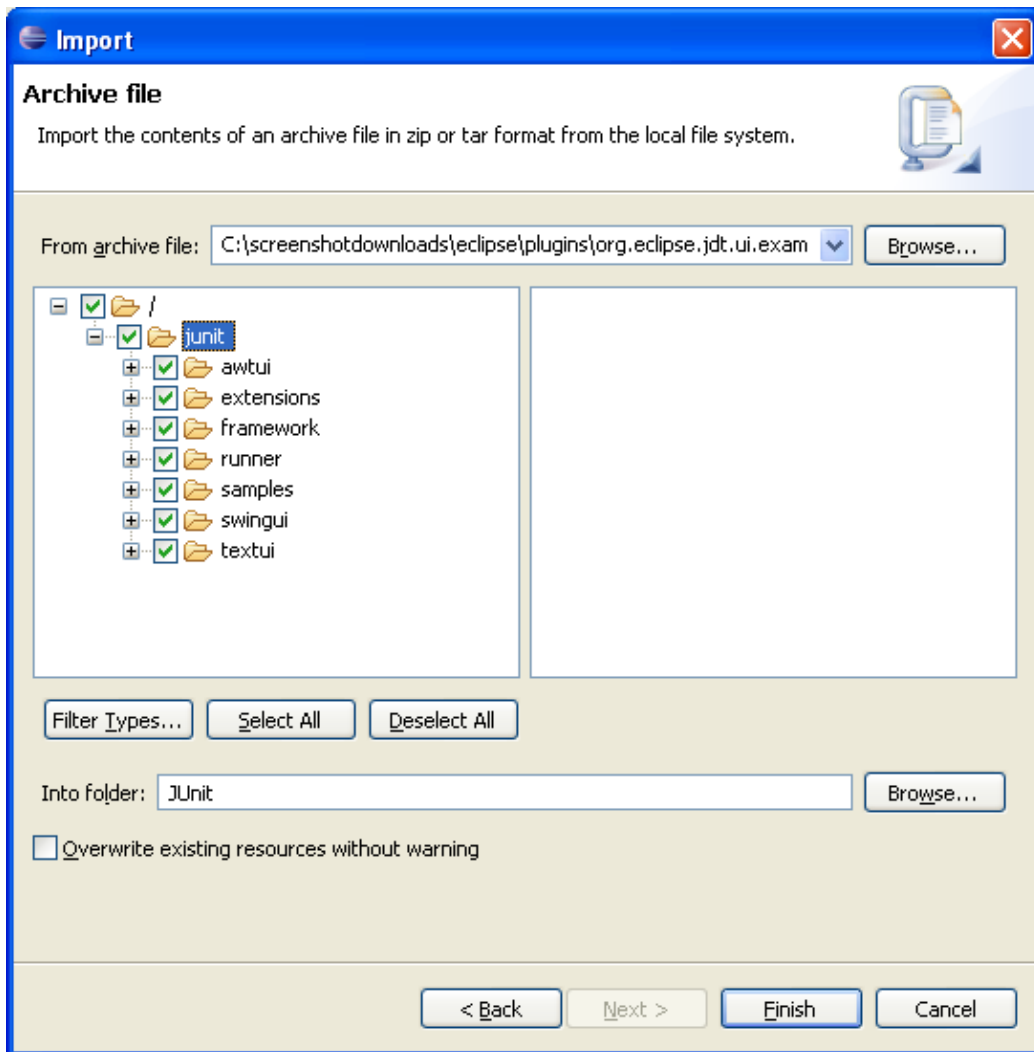
Creating the project

1. Inside Eclipse select the menu item **File > New > Project...** to open the *New Project* wizard
2. Select **Java Project** then click *Next* to start the *New Java Project* wizard:

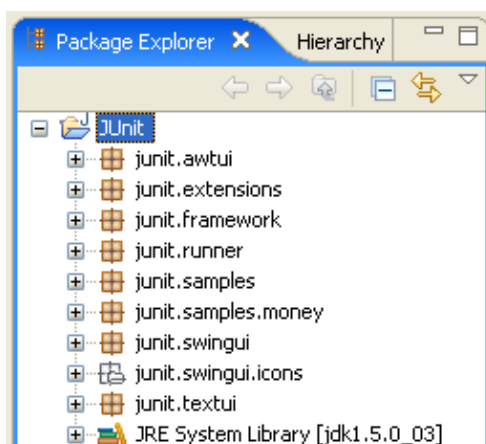


On this page, type "JUnit" in the **Project name** field. Make sure the **JDK compliance** is set to 1.4, and click **Finish**. (If you are interested in the new features in Eclipse supporting J2SE 5.0 development, see the companion guide "Getting Started with Java 5.0 development in Eclipse").

3. In the Package Explorer, make sure that the JUnit project is selected. Select the menu item **File > Import...**
4. Select **Archive file**, then click **Next**.
5. Click the **Browse** button next to the **Archive file** field and browse to select
 <Downloads>eclipse/plugins/org.eclipse.jdt.ui.examples.projects_3.0.0/archive/junit/junit381src.jar.
Note: This step assumes that you followed steps 1–3 in the **Getting the Sample Code** section above.
6. In the Import wizard, below the file tree click **Select All**. You can expand and select elements within the *junit* directory on the left pane to view the individual resources that you are importing on the right pane. **Note:** Do not deselect any of the resources in the junit directory at this time. You will need all of these resources in the tutorial.



7. Make sure that the JUnit project appears in the destination **Folder** field. Then click **Finish**. In the import progress indicator, notice that the imported resources are compiled as they are imported into the workbench. This is because the **Build automatically** option is checked on the Workbench preferences page. You will be prompted to overwrite the .classpath and .project files in the JUnit project. This is because the .classpath resource was created for you when you created the JUnit project. It is safe to overwrite these files.
8. In the Package Explorer view, expand the JUnit project to view the JUnit packages.



■ Related concepts

[Java projects](#)

[Java views](#)

■ Related tasks

[Working with build paths](#)

[Creating a new Java project](#)

[Using the Package Explorer](#)

■ Related reference

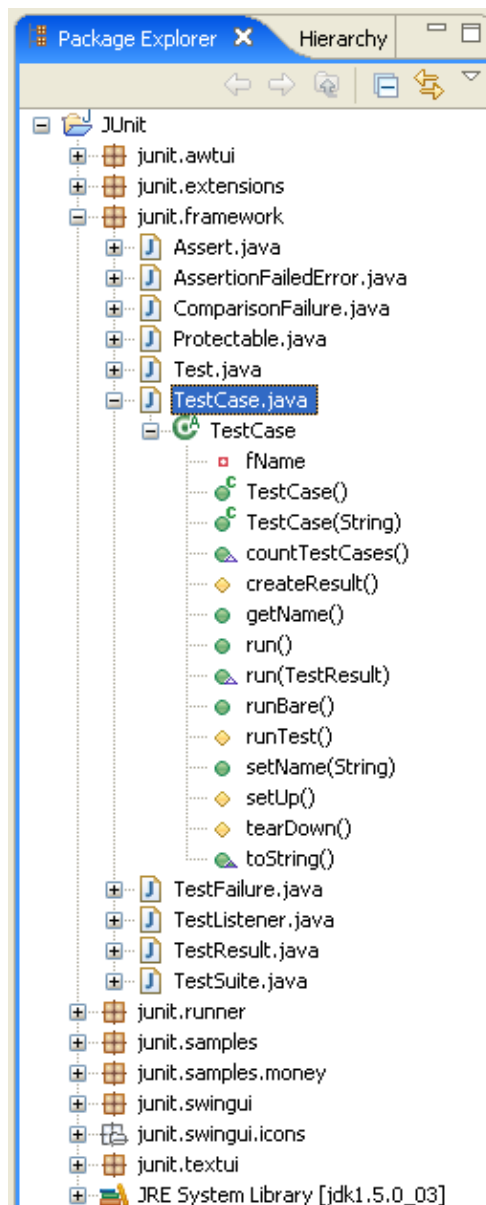
[New Java Project Wizard](#)

[Package Explorer View](#)

Browsing Java elements using the package explorer

In this section, you will browse Java elements within the JUnit project.

1. In the Package Explorer view, make sure the JUnit project is expanded so you can see its packages.
2. Expand the package `junit.framework` to see the Java files contained in the package.
3. Expand the Java file `TestCase.java`. Note that the Package Explorer shows Java-specific sub-elements of the source code file. The public type and its members (fields and methods) appear in the tree.



Related concepts

Java views

■ Related tasks

Using the Package Explorer

■ Related reference

Package Explorer View

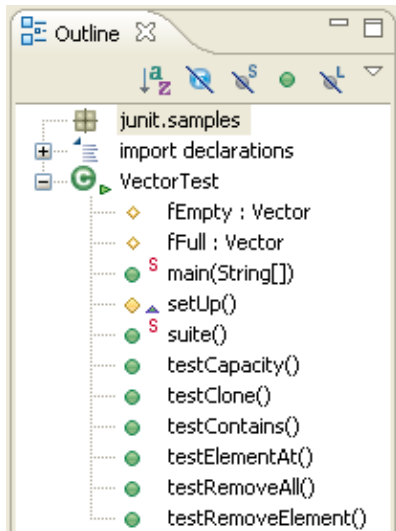
Opening a Java editor

In this section, you will learn how to open an editor for Java files. You will also learn about some of the basic Java editor features.

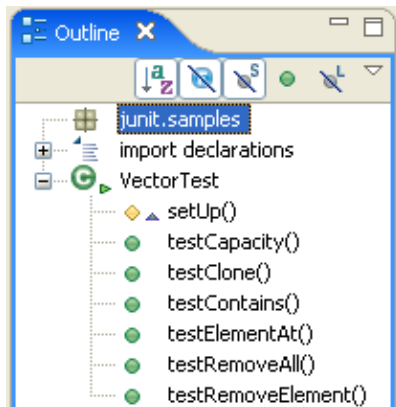
1. Expand the package *junit.samples* and select the file *VectorTest.java*. You can open *VectorTest.java* in the Java editor by double clicking on it. In general you can open a Java editor for Java files, types, methods and fields by simply double clicking on them. For example, to open the editor directly on the method *testClone* defined in *VectorTest.java*, expand the file in the Package Explorer and double click on the method.
2. Notice the syntax highlighting. Different kinds of elements in the Java source are rendered in unique colors. Examples of Java source elements that are rendered differently are:
 - ◆ Regular comments
 - ◆ Javadoc comments
 - ◆ Keywords
 - ◆ Strings.



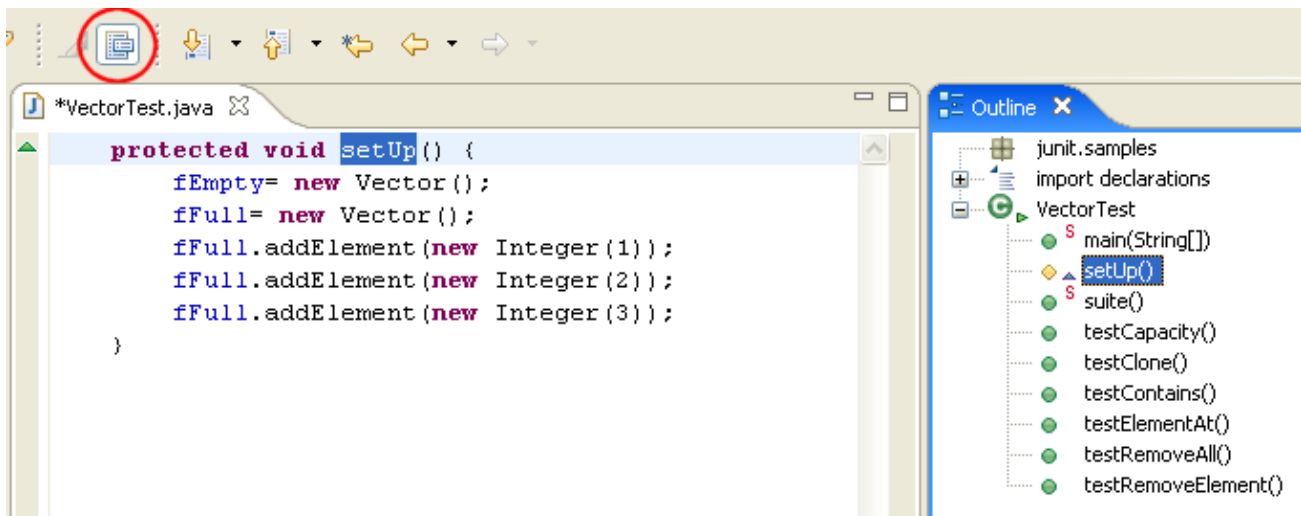
3. Look at the Outline view. It displays an outline of the Java file including the package declaration, import declarations, fields, types and methods. The Outline view uses icons to annotate Java elements. For example, icons indicate whether a Java element is static (S), abstract (A), or final (F). Different icons show you whether a method overrides a method from a base class (▲) or when it implements a method from an interface (△).



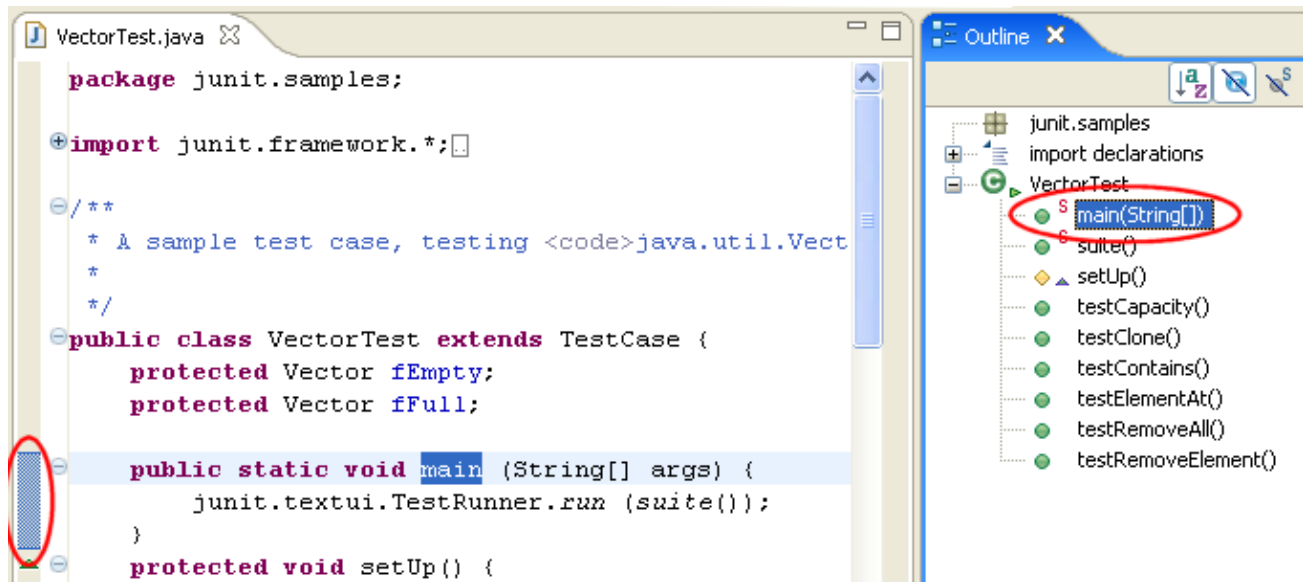
4. Toggle the **Hide Fields**, **Hide Static Members**, and **Hide Non-Public Members** buttons in the Outline view toolbar to filter the view's display. Before going to the next step make sure that **Hide Non-Public Members** button is not pressed.



5. You can edit source code by viewing the whole Java file, or you can narrow the view to a single Java element. The toolbar includes a button, **Show Source of Selected Element Only**, that will cause only the source code of the selected outline element to be displayed in the Java editor. In the example below, only the `setUp()` method is displayed.



6. Press the **Show Source of Selected Element Only** button again to see the whole Java file again. In the Outline view, select different elements and note that they are again displayed in a whole file view in the editor. The Outline view selection now contains a range indicator on the vertical ruler on the left border of the Java editor that indicates the range of the selected element.



Related concepts

[Java views](#)

[Java editor](#)

Related tasks

[Using the Java editor](#)

[Showing and hiding elements](#)

[Showing single elements or whole Java files](#)

[Sorting elements in Java views](#)

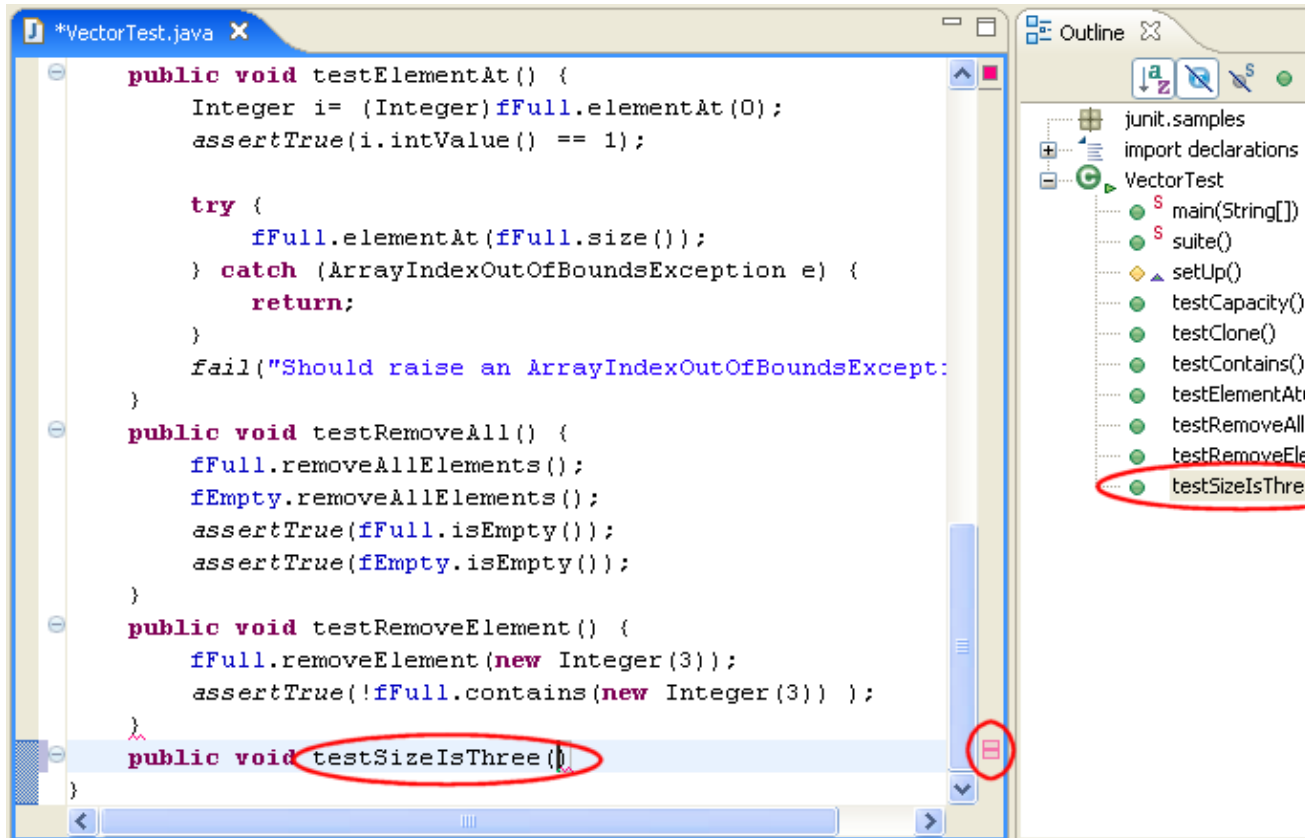
Related reference

[Java Outline View](#)

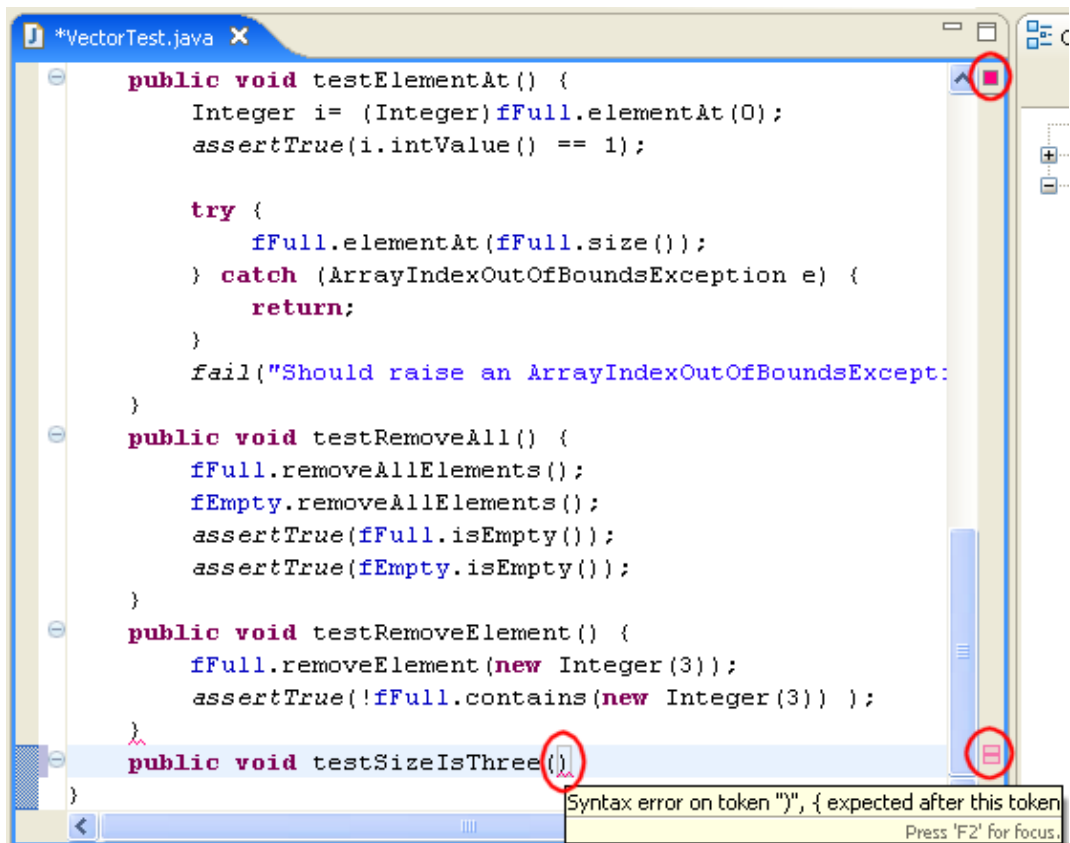
[Java Editor Preferences](#)

Adding new methods

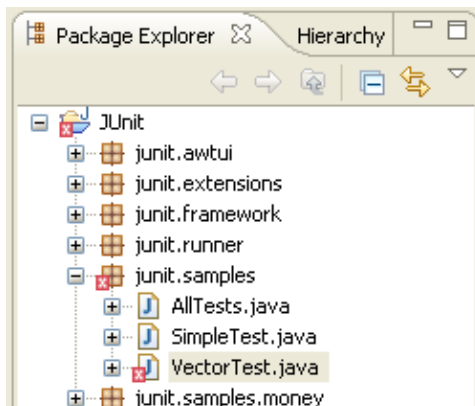
1. Start adding a method by typing the following at the end of the *VectorTest.java* file (but before the closing brackets of the type) in the Java editor:
`public void testSizeIsThree()`
As soon as you type the method name in the editor area, the new method appears at the bottom of the Outline view.



In addition, error annotations (red boxes) appear in the overview ruler positioned on the right hand side of the editor. These error annotations indicate that the compilation unit is currently not correct. If you hover over the second red box, a tool tip appears: *Unmatched bracket; Syntax error on token ")", { expected after this token*, which is correct since the method doesn't have a body yet. Note that error annotations in the editor's rulers are updated as you type. This behavior can be controlled via the **Report problems as you type** option located on the preference page **Java > Editor**.



2. Click the **Save** button. The compilation unit is compiled automatically and errors appear in the Package Explorer view, in the Problems view and on the vertical ruler (left hand side of the editor). In the Package Explorer view, the errors are propagated up to the project of the compilation unit containing the error.



3. Complete the new method by typing the following:

```

{
    assertTrue(fFull.size() == 3);
}

```

Note that the closing curly bracket has been auto inserted.

4. Save the file. Notice that the error indicators disappear since the missing bracket has been added.

Related concepts

Java editor

■ Related tasks

Using the Java editor

■ Related reference

Java Editor Preferences

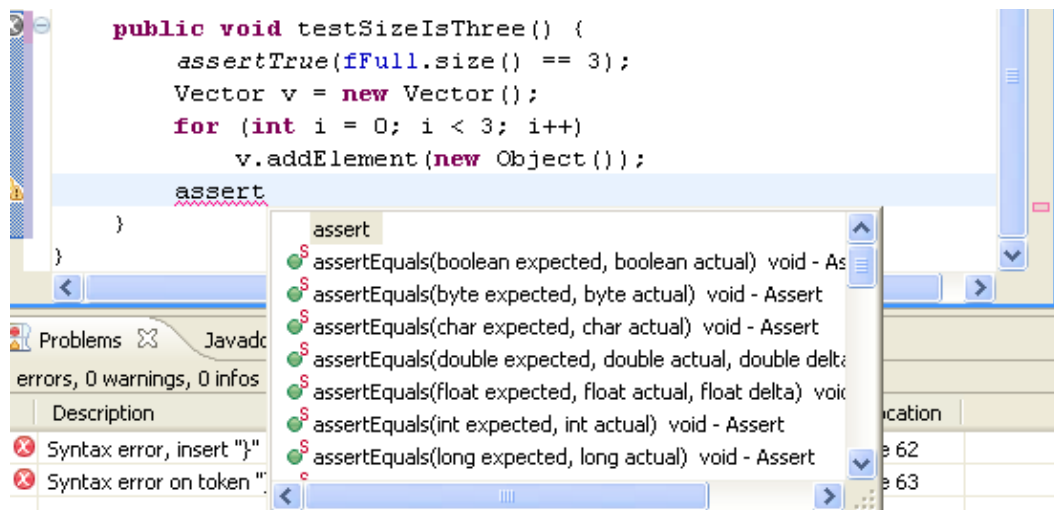
Using content assist

In this section you will use *content assist* to finish writing a new method. Open *junit.samples.VectorTest.java* file in the Java editor if you do not already have it open and select the `testSizeIsThree()` method in the Outline view. If the file doesn't contain such a method see [Adding new methods](#) for instructions on how to add this method.

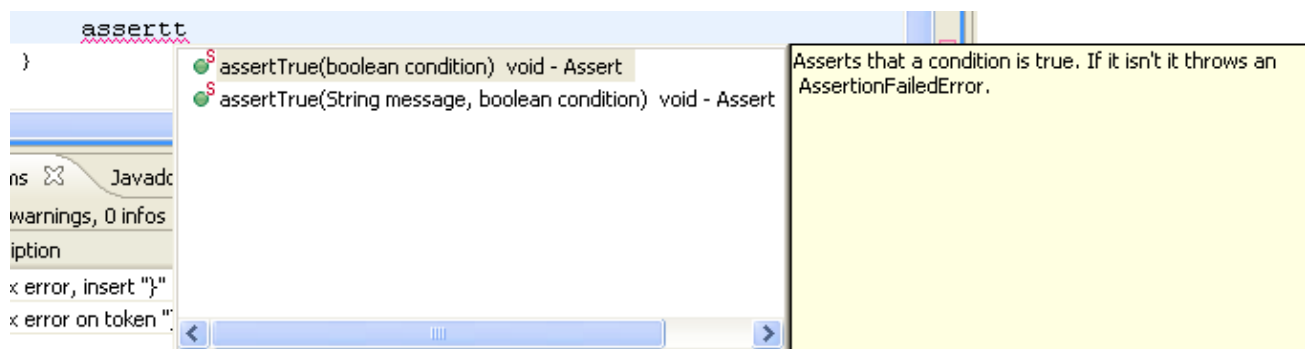
1. Add the following lines to the end of the method. When you type the `.` after `v`, Eclipse will show a list of possible completions. Notice how the options are narrowed down as you continue to type.

```
Vector v = new Vector();
for (int i=0; i<3; i++)
    v.addElement(new Object());
assert
```

2. With your cursor at the end of the word `assert`, press **Ctrl+Space** to activate content assist. The content assist window with a list of proposals will appear. Scroll the list to see the available choices.



3. With the content assist window still active, type the letter `'t'` in the source code after `assert` (with no space in between). The list is narrowed and only shows entries starting with `'assert'`. Single-click various items in the list to view any available Javadoc help for each item.



4. Select `assertTrue(boolean)` from the list and press **Enter**. The code for the `assertTrue(boolean)` method is inserted.
5. Complete the line so that it reads as follows:

Basic tutorial

```
assertTrue(v.size() == fFull.size());
```

6. Save the file.

■ Related concepts

[Java editor](#)

■ Related tasks

[Using the Java editor](#)

[Using content assist](#)

■ Related reference

[Java Content Assist](#)

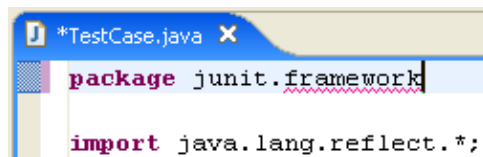
[Java Editor Preferences](#)

Identifying problems in your code

In this section, you will review the different indicators for identifying problems in your code.

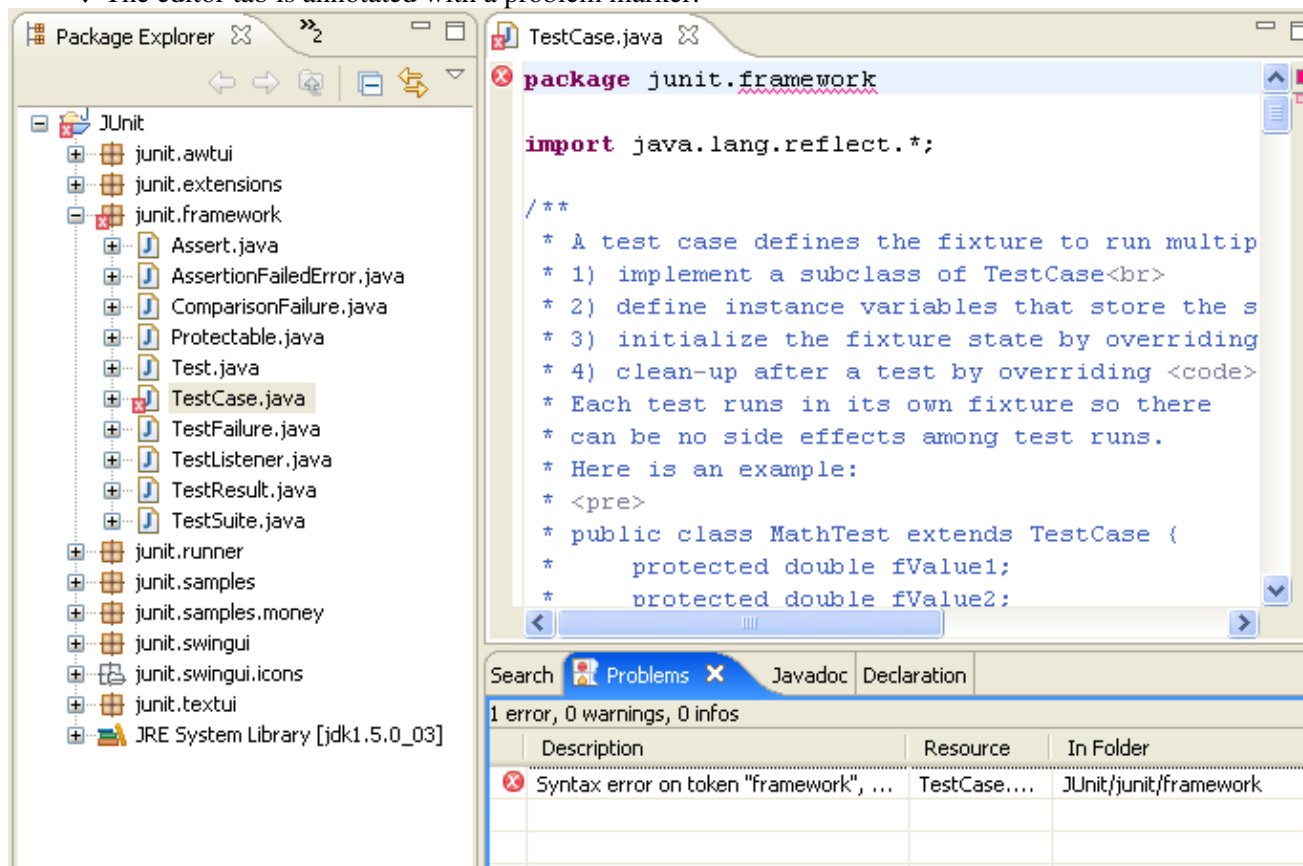
Build problems are displayed in the Problems view and annotated in the vertical ruler of your source code.

1. Open *junit.framework.TestCase.java* in the editor from the Package Explorer view.
2. Add a syntax error by deleting the semicolon at the end of the package declaration in the source code.



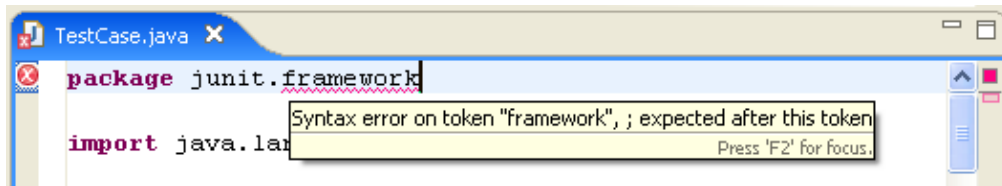
```
package junit.framework  
  
import java.lang.reflect.*;
```

3. Click the **Save** button. The project is rebuilt and the problem is indicated in several ways:
 - ◆ In the Problems view, the problems are listed,
 - ◆ In the Package Explorer view, the Type Hierarchy or the Outline view, problem ticks appear on the affected Java elements and their parent elements,
 - ◆ In the editor's vertical ruler, a problem marker is displayed near the affected line,
 - ◆ Squiggly lines appear under the word which might have caused the error, and
 - ◆ The editor tab is annotated with a problem marker.

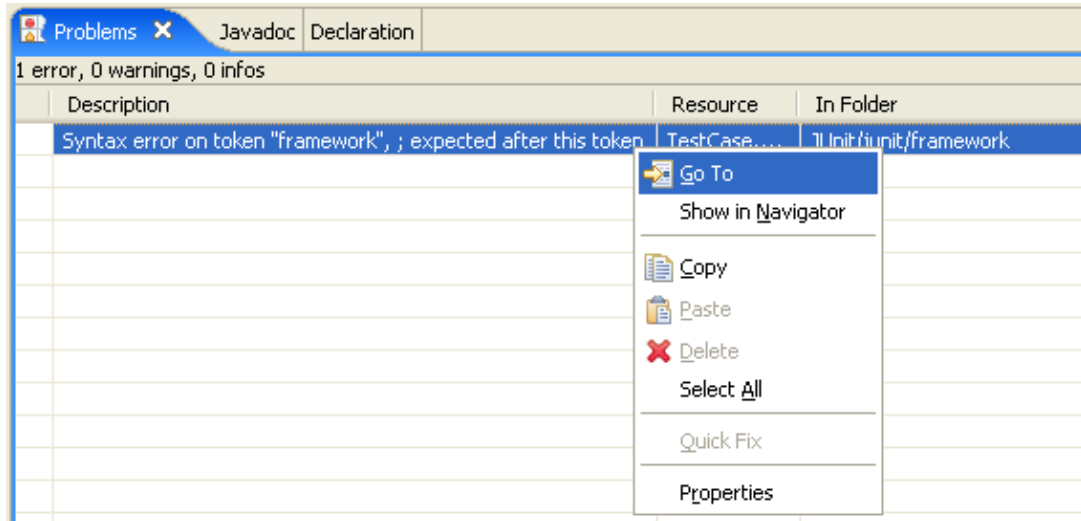


4. You can hover over the problem marker in the vertical ruler to view a description of the problem.

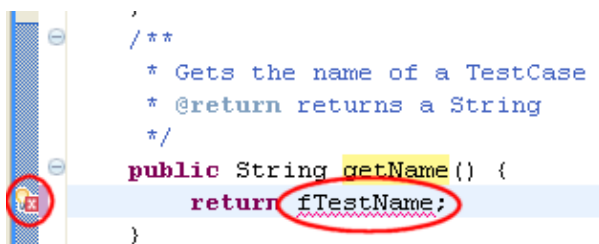
Basic tutorial



5. Click the **Close** ("X") button on the editor's tab to close the editor.
6. In the Problems view, select a problem in the list. Open its context menu and select **Go To**. The file is opened in the editor at the location of the problem.

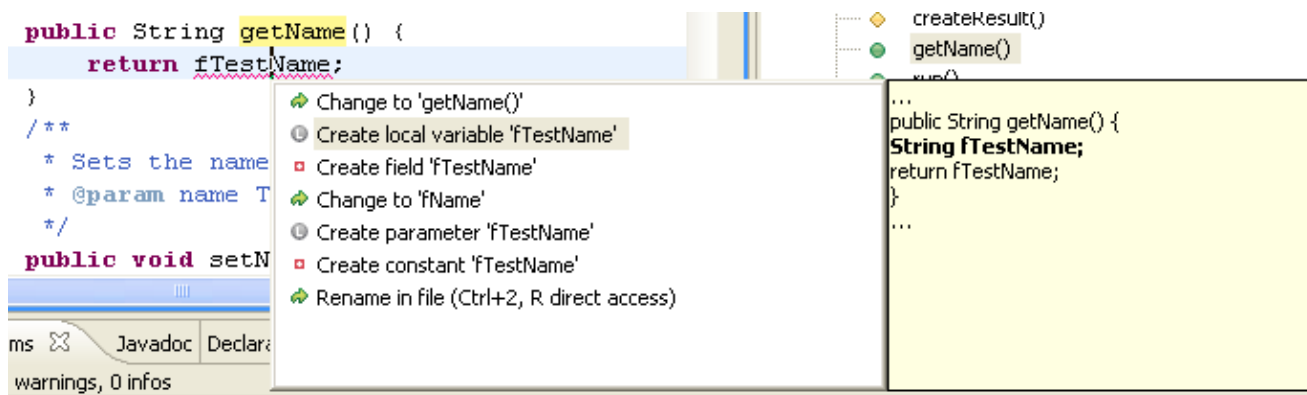


7. Correct the problem in the editor by adding the semicolon. Click the **Save** button. The project is rebuilt and the problem indicators disappear.
8. In the Outline view, select the method `getName()`. The editor will scroll to this method.
9. On the first line of the method change the returned variable `fName` to `fTestName`. While you type, a problem highlight underline appears on `fTestName`, to indicate a problem. Hovering over the highlighted problem will display a description of the problem.
10. On the marker bar a light bulb marker appears. The light bulb signals that correction proposals are available for this problem.



11. Click to place the cursor onto the highlighted error, and choose **Quick Fix** from the Edit menu bar. You can also press **Ctrl+1** or left click the light bulb. A selection dialog appears with possible corrections.

Basic tutorial



12. Select 'Change to fName' to fix the problem. The problem highlight line will disappear as the correction is applied.
13. Close the file without saving.
14. You can configure how problems are indicated on the **Window > Preferences > General > Editors > Text Editors > Annotations** page.

■ Related concepts

[Java editor](#)

[Java views](#)

[Java builder](#)

■ Related tasks

[Using the Java editor](#)

[Viewing documentation and information](#)

[Using quick fix](#)

■ Related reference

[Editor preference page](#)

[Quick Fix](#)

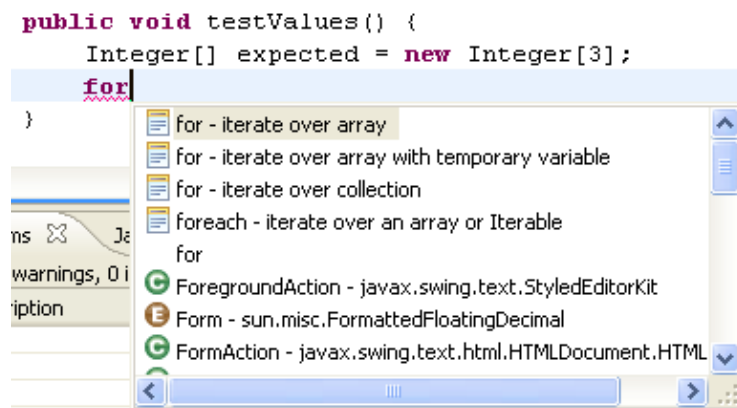
Using code templates

In this section you will use content assist to fill in a template for a common loop structure. Open *junit.samples.VectorTest.java* file in the Java editor if you do not already have it open.

1. Start adding a new method by typing the following:

```
public void testValues() {  
    Integer[] expected= new Integer[3];  
    for
```

2. With the cursor at the end of `for`, press **Ctrl+Space** to enable content assist. You will see a list of common templates for "for" loops. When you single-click a template, you'll see the code for the template in its help message. Note that the local array name is guessed automatically.



3. Choose the `for - iterate over array` entry and press **Enter** to confirm the template. The template will be inserted in your source code.

```
public void testValues() {  
    Integer[] expected = new Integer[3];  
    for (int i = 0; i < expected.length; i++) {  
          
    }  
}
```

4. Next we change the name of the index variable from `i` to `e`. To do so simply press **e**, as the index variable is automatically selected. Observe that the name of the index variable changes at all places. When inserting a template all references to the same variable are connected to each other. So changing one changes all the other values as well.

```
public void testValues() {  
    Integer[] expected = new Integer[3];  
    for (int e = 0; e < expected.length; e++) {  
          
    }  
}
```

5. Pressing the **tab** key moves the cursor to the next variable of the code template. This is the array *expected*.

Basic tutorial

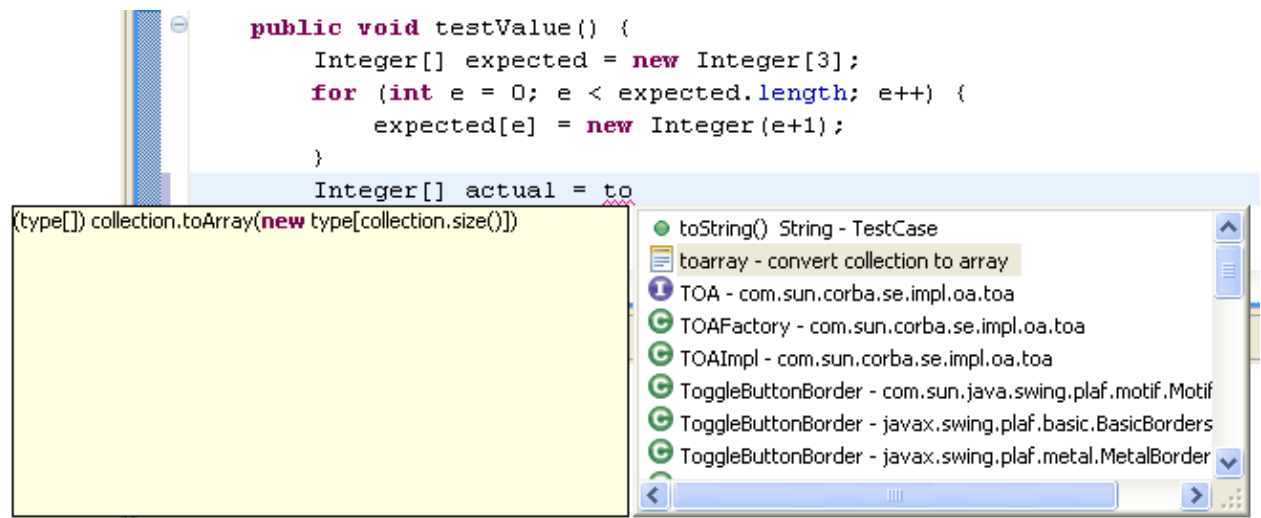
```
public void testValues() {  
    Integer[] expected = new Integer[3];  
    for (int e = 0; e < expected.length; e++) {  
    }  
}
```

Since we don't want to change the name (it was guessed right by the template) we press tab again, which leaves the template since there aren't any variables left to edit.

6. Complete the for loop as follows:

```
for (int e = 0; e < expected.length; e++) {  
    expected[e] = new Integer(e + 1);  
}  
Integer[] actual = to
```

7. With the cursor at the end of to, press **Ctrl+Space** to enable content assist. Pick toArray - convert collection to array and press **Enter** to confirm the selection (or double-click the selection).



The template is inserted in the editor and type is highlighted and selected.

```
public void testValues() {  
    Integer[] expected = new Integer[3];  
    for (int e = 0; e < expected.length; e++) {  
        expected[e] = new Integer(e + 1);  
    }  
    Integer[] actual = (type[]) collection.toArray(new type[collection.size()])  
}
```

8. Overwrite the selection by typing Integer. The type of array constructor changes when you change the selection.
9. Press **Tab** to move the selection to collection and overwrite it by typing fFull.

Basic tutorial

```
public void testValues() {  
    Integer[] expected = new Integer[3];  
    for (int e = 0; e < expected.length; e++) {  
        expected[e] = new Integer(e + 1);  
    }  
    Integer[] actual = (Integer[]) fFull.toArray(new Integer[fFull.size()]);  
}
```

10. Add the following lines of code to complete the method:

```
assertEquals(expected.length, actual.length);  
for (int i = 0; i < actual.length; i++)  
    assertEquals(expected[i], actual[i]);
```

11. Save the file.

Related concepts

[Java editor](#)

[Templates](#)

Related tasks

[Using the Java editor](#)

[Using templates](#)

Related reference

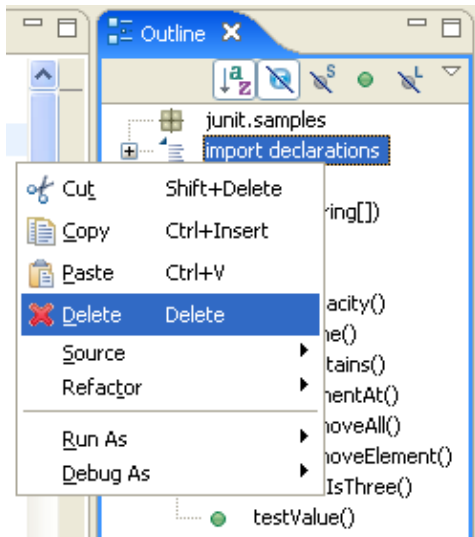
[Templates Preferences](#)

[Java Editor Preferences](#)

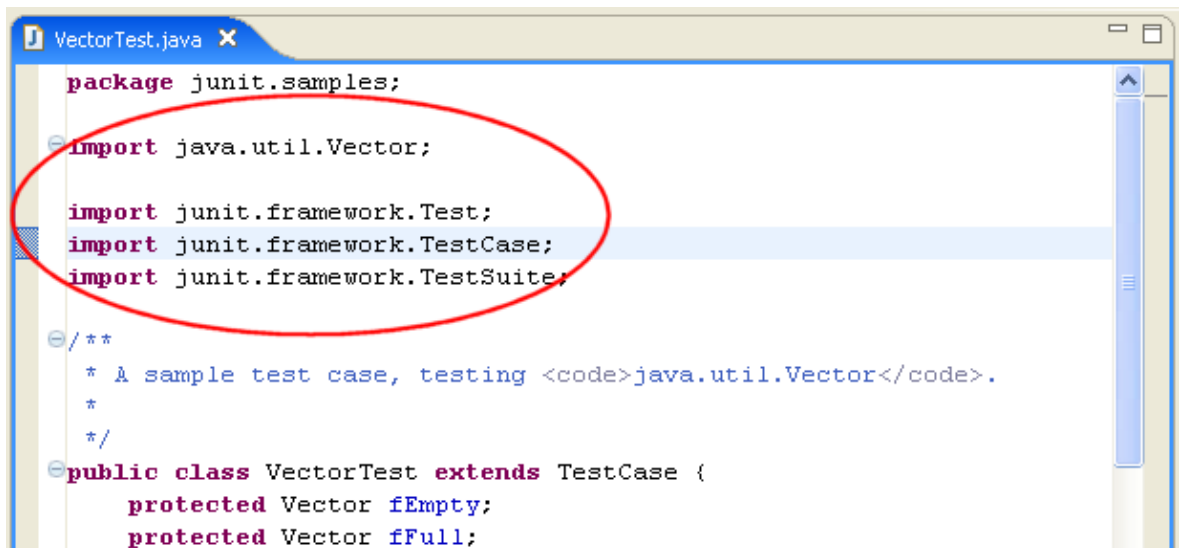
Organizing import statements

In this section you will organize the import declarations in your source code. Open `junit.samples.VectorTest.java` file in the Java editor if you do not already have it open.

1. Delete the import declarations by selecting them in the Outline view and selecting **Delete** from the context menu. Confirm the resulting dialog with **Yes**. You will see numerous compiler warnings in the vertical ruler since the types used in the method are no longer imported.



2. From the context menu in the editor, select **Source > Organize Imports**. The required import statements are added to the beginning of your code below the package declaration.



You can also choose **Organize Imports** from the context menu of the import declarations in the Outline view.

Note: You can specify the order of the import declarations in preferences **Window > Preferences > Java > Code Style > Organize Imports**.

3. Save the file.

- Related concepts

Java editor

- Related tasks

Managing import statements

- Related reference

Organize Imports Preferences

Using the local history

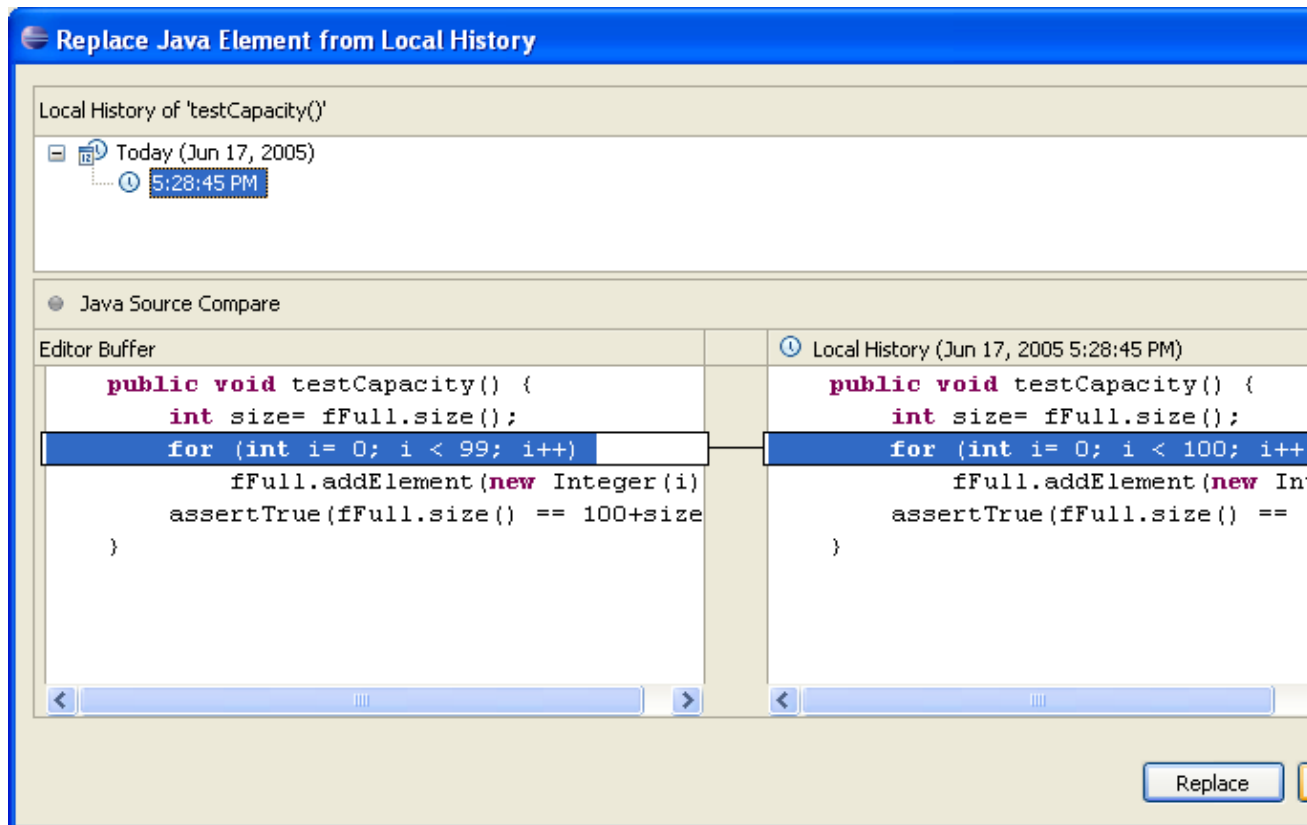
In this section, you will use the local history feature to switch to a previously saved version of an individual Java element.

1. Open *junit.samples.VectorTest.java* file in the Java editor and select the method *testCapacity()* in the Outline view.
2. Change the content of the method so that the 'for' statements reads as:

```
for (int i= 0; i < 99; i++)
```

Save the file by pressing Ctrl+S.

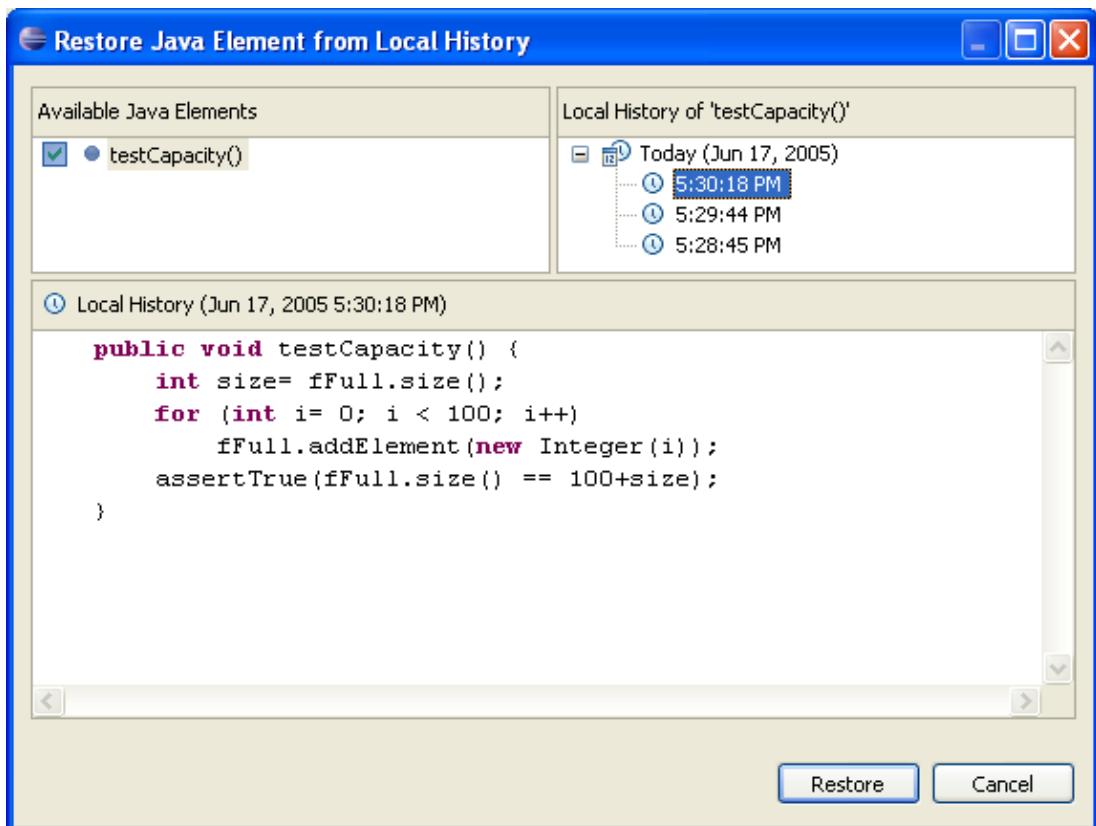
3. In the Outline view, select the method *testCapacity()*, and from its context menu, select **Replace With > Element from Local History**.
4. In the **Replace Java Element from Local History** dialog, the Local History list shows the various saved states of the method. The Java Source Compare pane shows details of the differences between the selected history resource and the existing workbench resource.



5. In the top pane, select the previous version, and click the **Replace button**. In the Java editor, the method is replaced with the selected history version.
6. Save the file.
7. Beside replacing a method's version with a previous one you can also restore Java elements that were deleted. Again, select the method *testCapacity()* in the Outline view, and from its context menu, select **Delete**. Confirm the resulting dialog with **Yes** and save the file.
8. Now select the type *VectorTest* in the Outline view, and from its context menu, select **Restore from Local History....** Select and check the method *testCapacity()* in the Available Java Elements pane. As

Basic tutorial

before, the Local History pane shows the versions saved in the local history.



9. In the Local History pane, select the earlier working version and then click **Restore**.
10. Press **Ctrl+S** to save the file.

■ Related concepts

[Java editor](#)

■ Related tasks

[Using the Java editor](#)

[Using the local history](#)

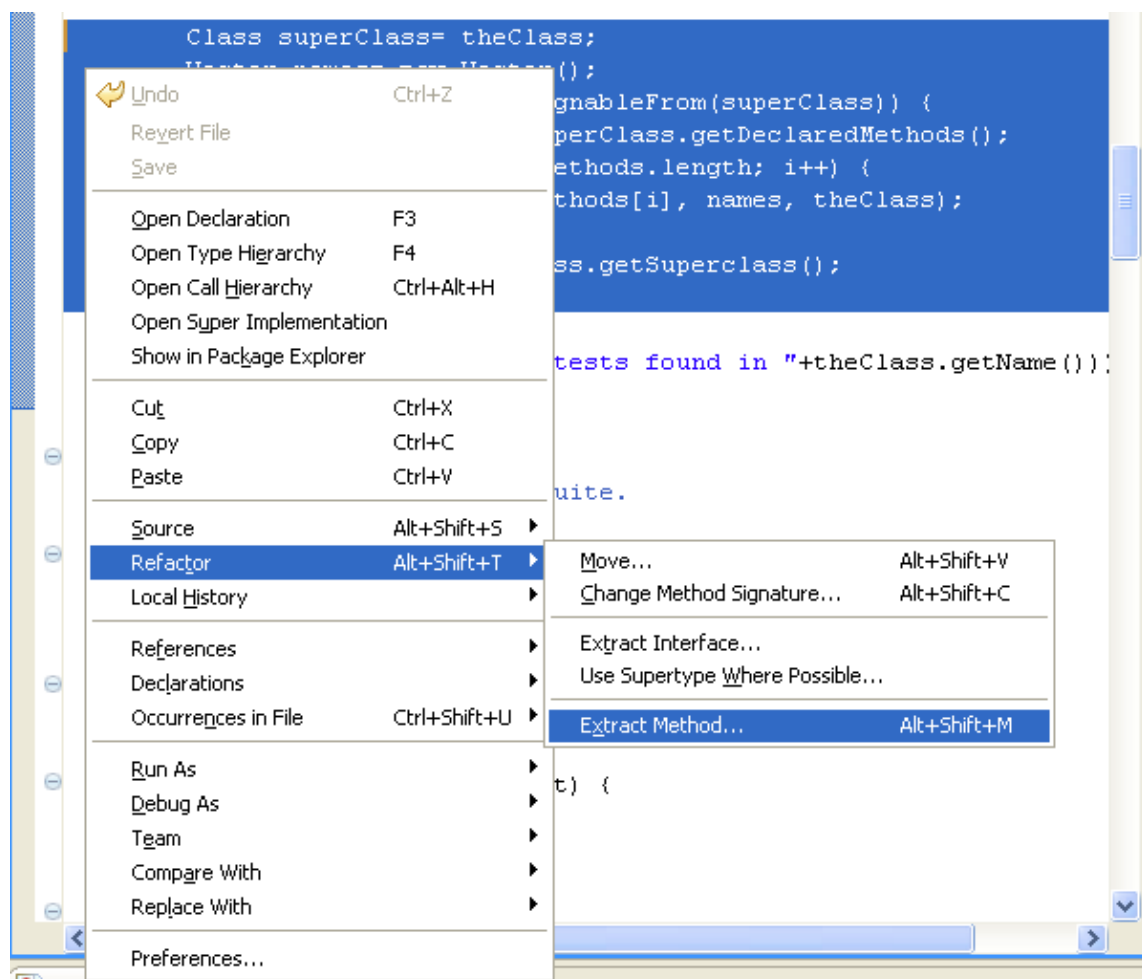
Extracting a new method

In this section, you will improve the code of the constructor of *junit.framework.TestSuite*. To make the intent of the code clearer, you will extract the code that collects test cases from base classes into a new method called *collectTestMethods*.

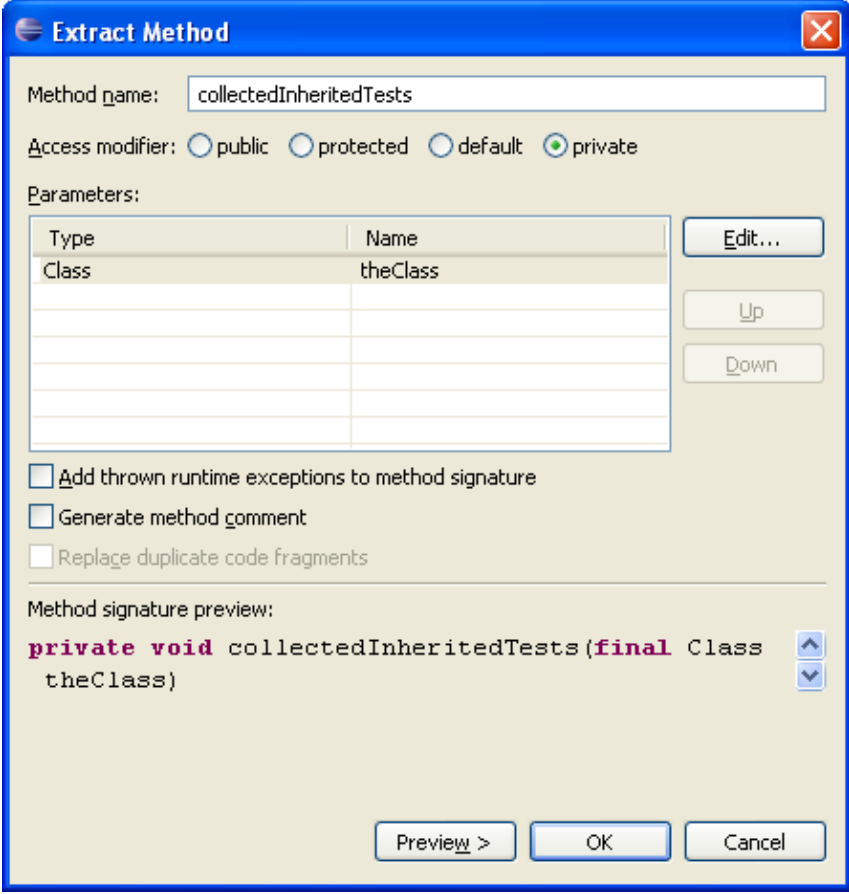
1. In the *junit.framework.TestSuite.java* file, select the following range of code inside the *TestSuite(Class)* constructor:

```
Class superClass= theClass;
Vector names= new Vector();
while(Test.class.isAssignableFrom(superClass)) {
    Method[] methods= superClass.getDeclaredMethods();
    for (int i= 0; i < methods.length; i++) {
        addTestMethod(methods[i],names, constructor);
    }
    superClass= superClass.getSuperclass();
}
```

2. From the selection's context menu in the editor, select **Refactor > Extract Method...**.



3. In the **Method Name** field, type *collectInheritedTests*.



The 'Extract Method' dialog box is shown with the following settings:

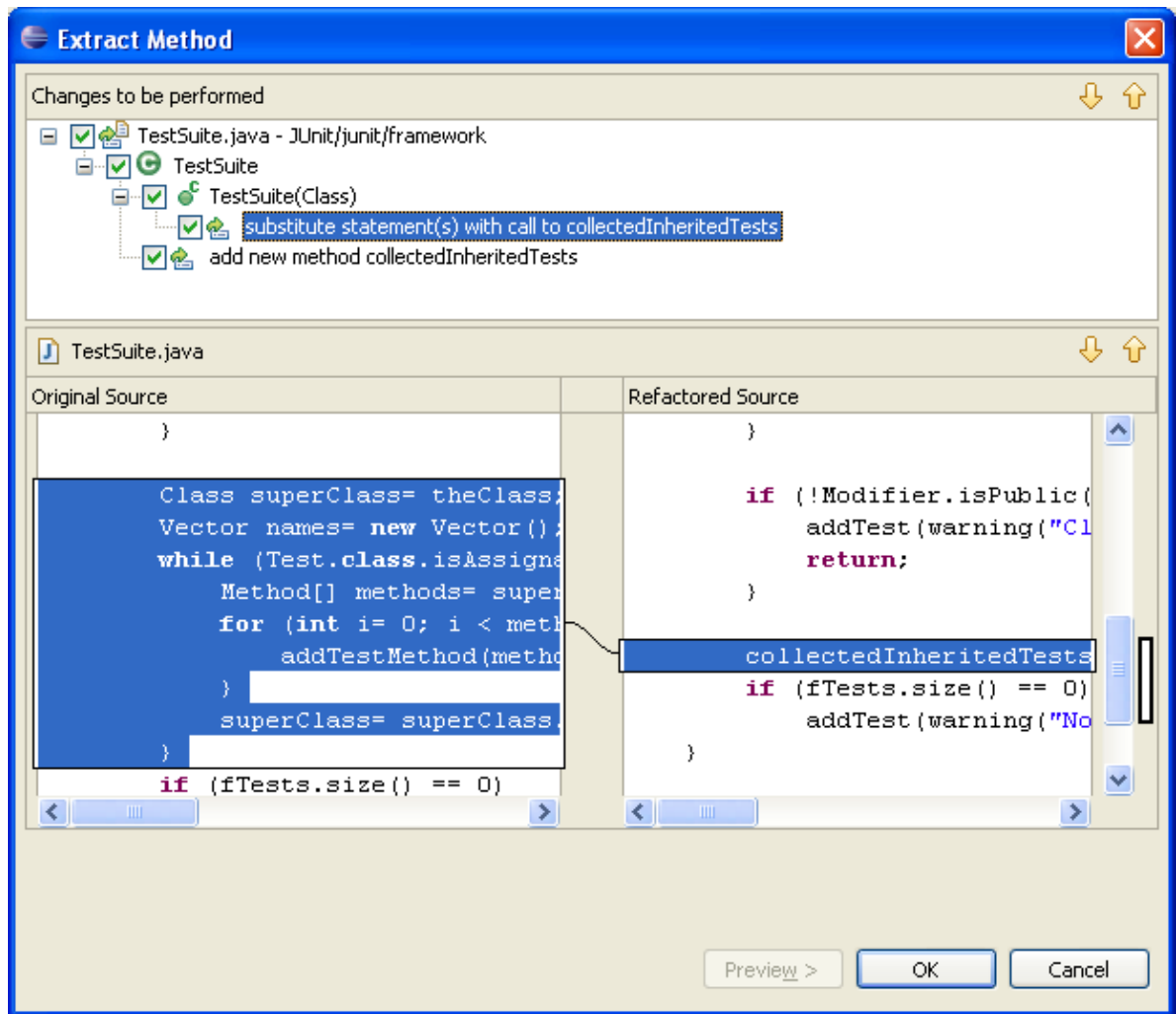
- Method name: `collectedInheritedTests`
- Access modifier: ☒ private
- Parameters table:

Type	Name
Class	theClass
- ☐ Add thrown runtime exceptions to method signature
- ☐ Generate method comment
- ☐ Replace duplicate code fragments
- Method signature preview:

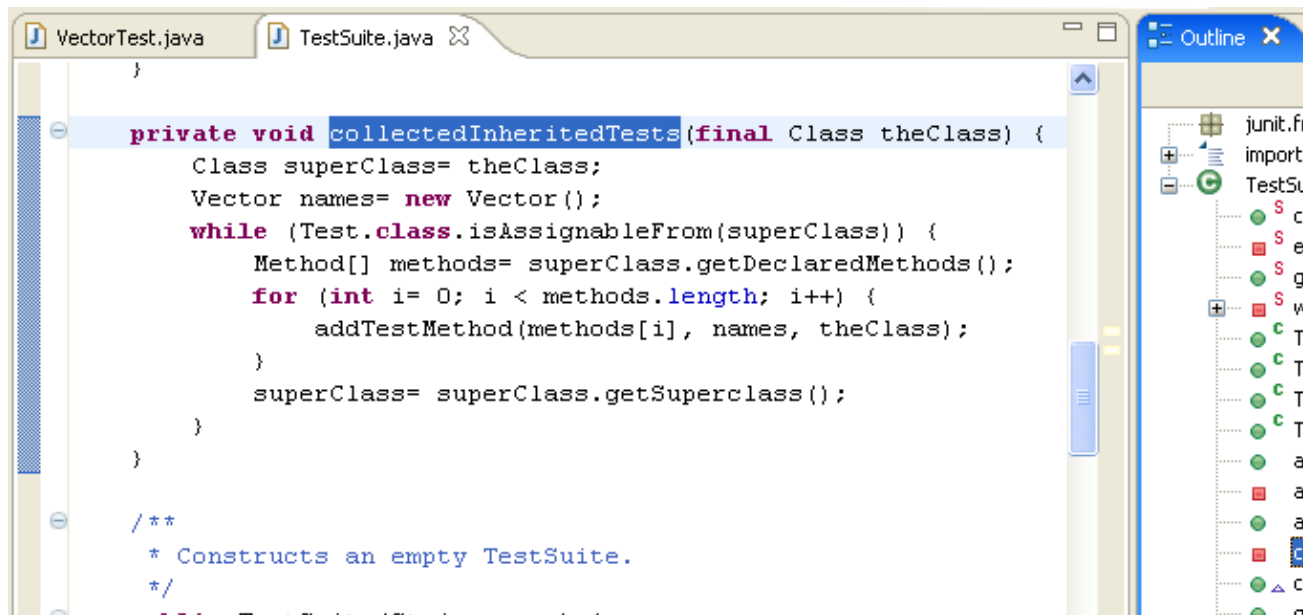
```
private void collectedInheritedTests (final Class theClass)
```

Buttons at the bottom: **Preview >**, **OK**, **Cancel**.

4. To preview the changes, press **Preview >**. The preview page displays the changes that will be made. Press **OK** to extract the method.



5. Go to the extracted method by selecting it in the Outline view.



Related concepts

[Java editor](#)
[Refactoring support](#)

■ Related tasks

[Using the Java editor](#)
[Refactoring](#)
[Refactoring with preview](#)

■ Related reference

[Extract Method Errors](#)
[Java Preferences](#)

Creating a Java class

In this section, you will create a new Java class and add methods using code generation actions.

1. In the Package Explorer view, select the JUnit project. Click the **New Java Package** button in the toolbar, or select **New > Package** from the project's context menu .
2. In the **Name** field, type *test* as the name for the new package. Then click **Finish**.
3. In the Package Explorer view, select the new *test* package and click the **New Java Class** button in the toolbar.
4. Make sure that *JUnit* appears in the **Source Folder** field and that *test* appears in the **Package** field. In the **Name** field, type *MyTestCase*.

New Java Class

Java Class
Create a new Java class.

Source folder: JUnit Browse...

Package: test Browse...

☐ Enclosing type: junit.framework.MyTestCase Browse...

Name: MyTestCase

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass: java.lang.Object Browse...

Interfaces: Add... Remove

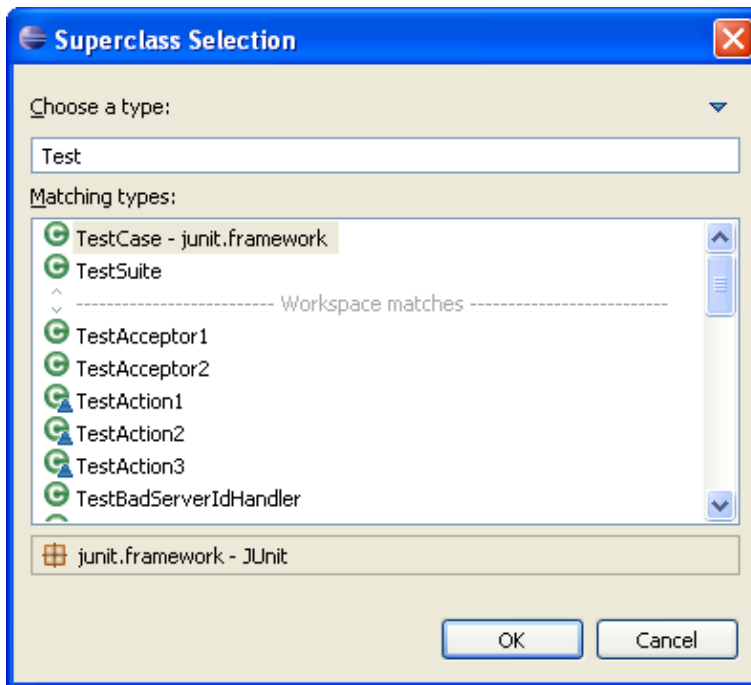
Which method stubs would you like to create?

☐ public static void main(String[] args)
☐ Constructors from superclass
☒ Inherited abstract methods

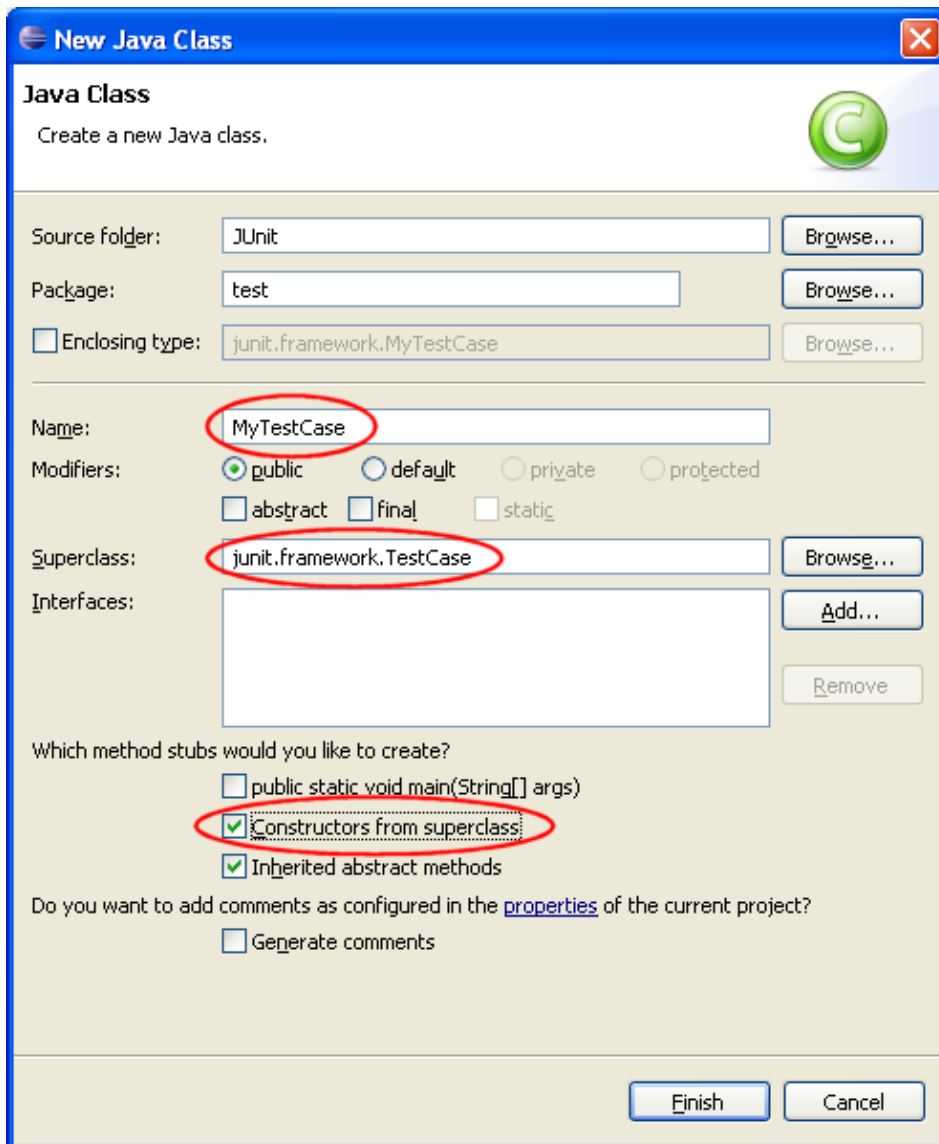
Do you want to add comments as configured in the [properties](#) of the current project?
☐ Generate comments

Finish Cancel

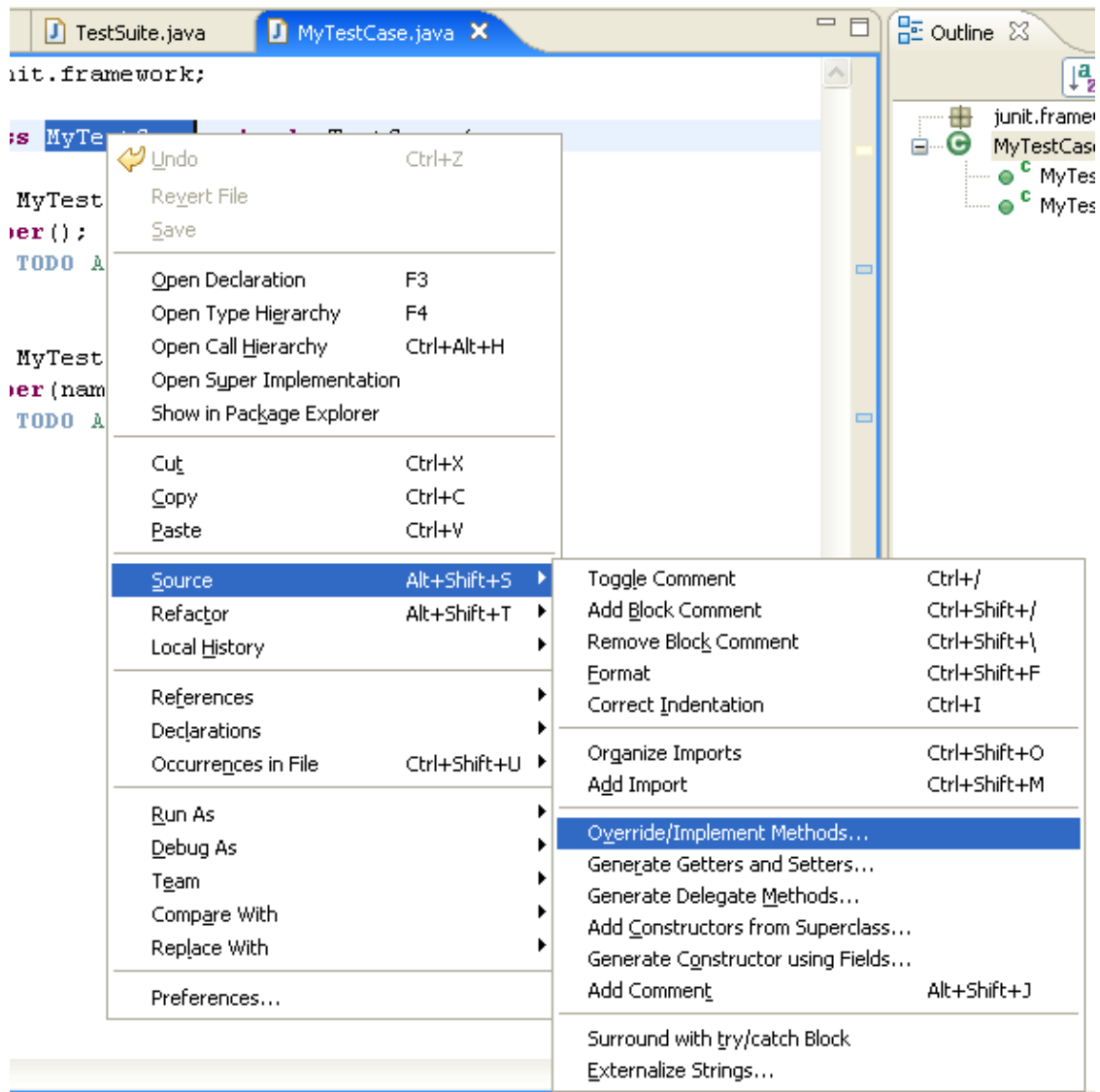
5. Click the **Browse** button next to the **Superclass** field.
6. In the **Choose a type** field in the Superclass Selection dialog, type *Test* to narrow the list of available superclasses.



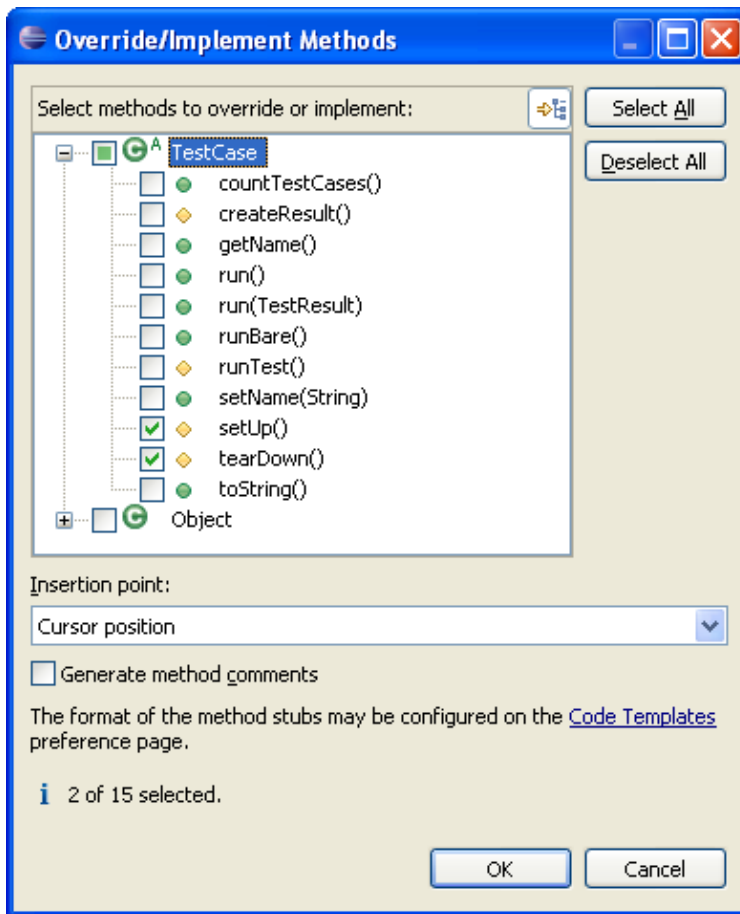
7. Select the *TestCase* class and click **OK**.
8. Select the checkbox for ***Constructors from superclass***.
9. Click ***Finish*** to create the new class.



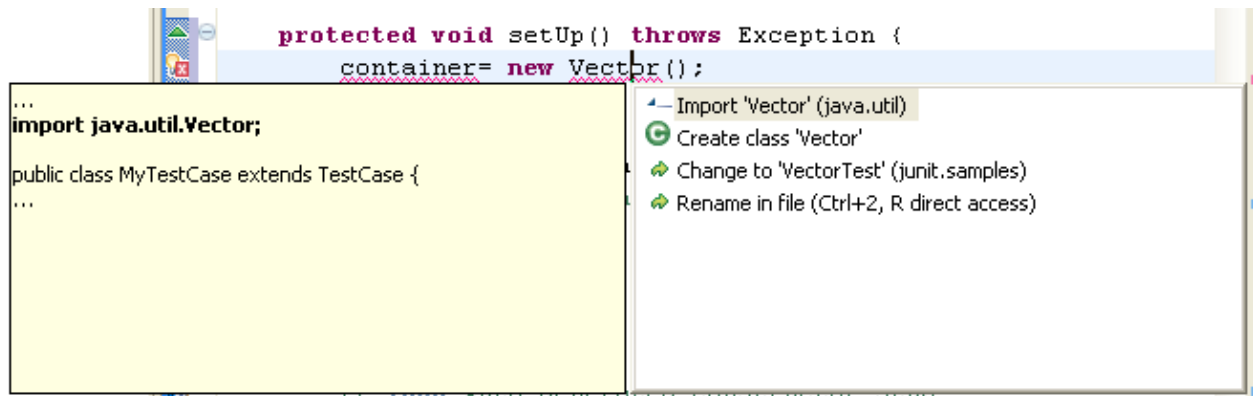
10. The new file is opened in the editor. It contains the new class, the constructor and comments. You can select options for the creation and configuration of generated comments in the Java preferences (*Window > Preferences > Java > Code Style > Code Templates*).
11. In the Outline view select the new class *MyTestCase*. Open the context menu and select *Source > Override/Implement Methods...*



12. In the Override Methods dialog, check *setUp()* and *tearDown()* and click **OK**. Two methods are added to the class.



13. Change the body of `setUp()` to `container= new Vector();`
14. `container` and `Vector` are underlined with a problem highlight line as they cannot be resolved. A light bulb appears on the marker bar. Set the cursor inside `Vector` and press **Ctrl+1** (or use **Edit > Quick Fix** from the menu bar). Choose **Import 'Vector' (java.util)**. This adds the missing import declaration.

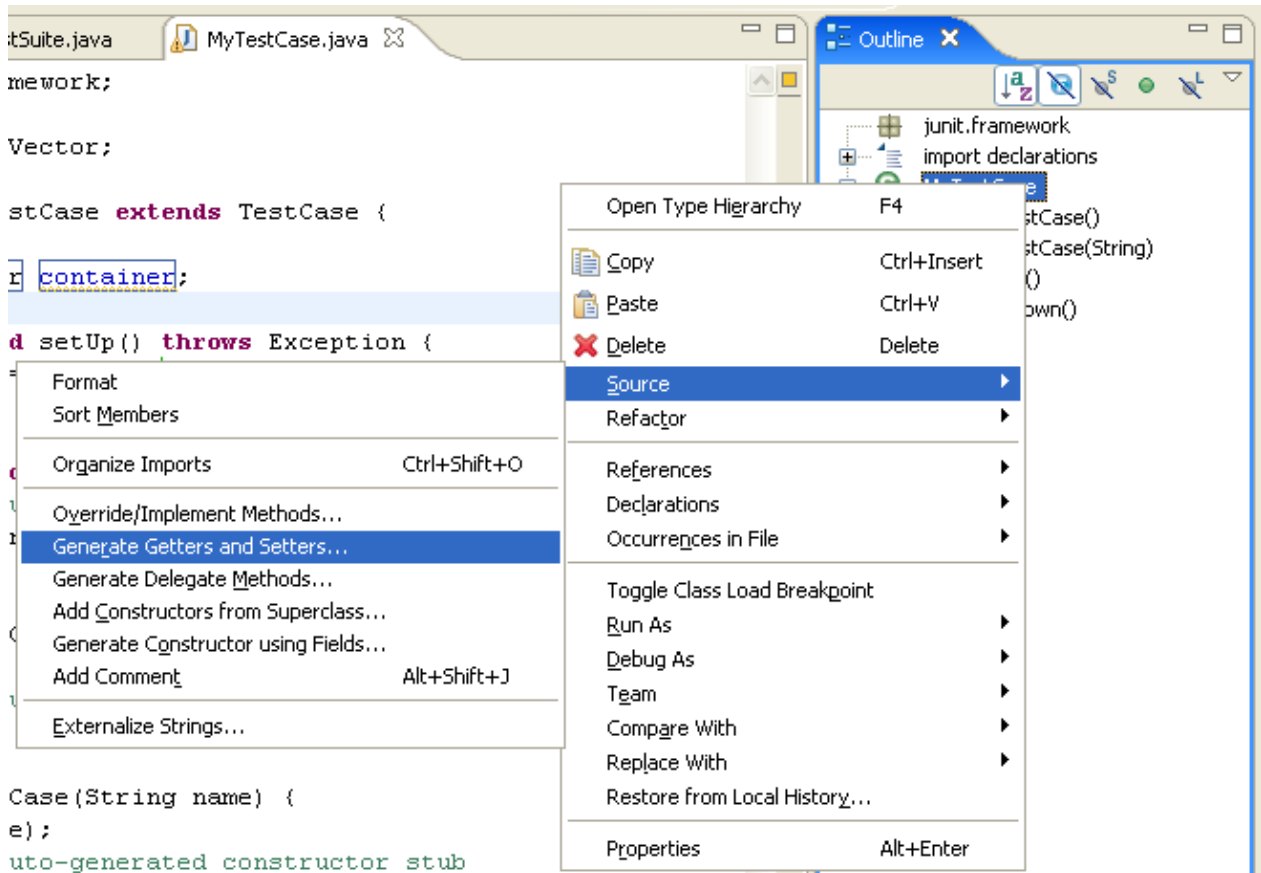


Set the cursor inside `container` and press **Ctrl+1**. Choose **Create field 'container'** to add the new field.

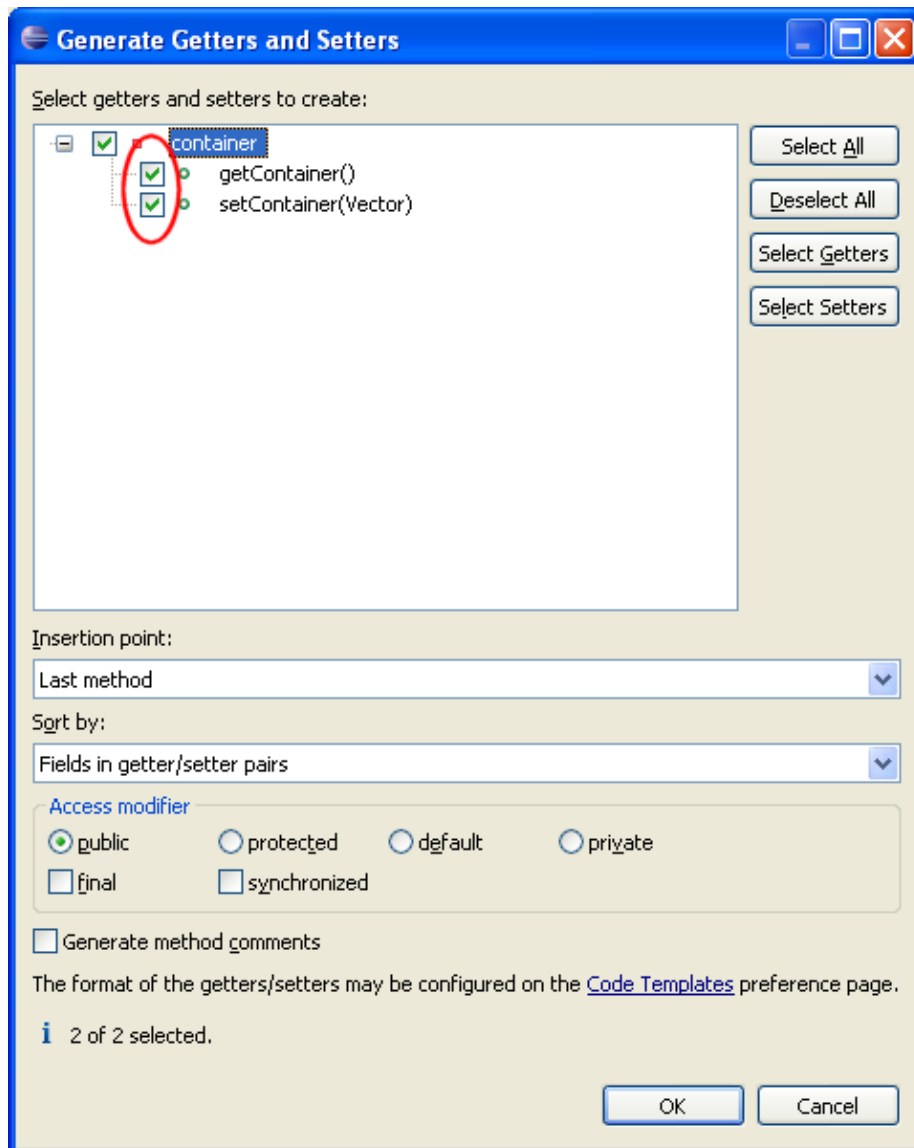
Basic tutorial



15. In the Outline view, select the class *MyTestCase*. Open the context menu and select **Source > Generate Getters and Setters...**



16. The Generate Getter and Setter dialog suggests that you create the methods `getContainer` and `setContainer`. Select both and click **OK**. A getter and setter method for the field `container` are added.



17. Save the file.
18. The formatting of generated code can be configured in **Window > Preferences > Java > Code Style > Formatter**. If you use a prefix or suffix for field names (e.g. fContainer), you can specify this in **Window > Preferences > Java > Code Style** so that generated getters and setters will suggest method names without the prefix or suffix.

● Related concepts

Java views

Java editor

● Related tasks

Using quick fix

Creating Java elements

Generating getters and setters

■ Related reference

[New Java Class wizard](#)

[Source actions](#)

[Quick Fix](#)

[Override Methods dialog](#)

[Generate Getter and Setter dialog](#)

[Code formatter preference page](#)

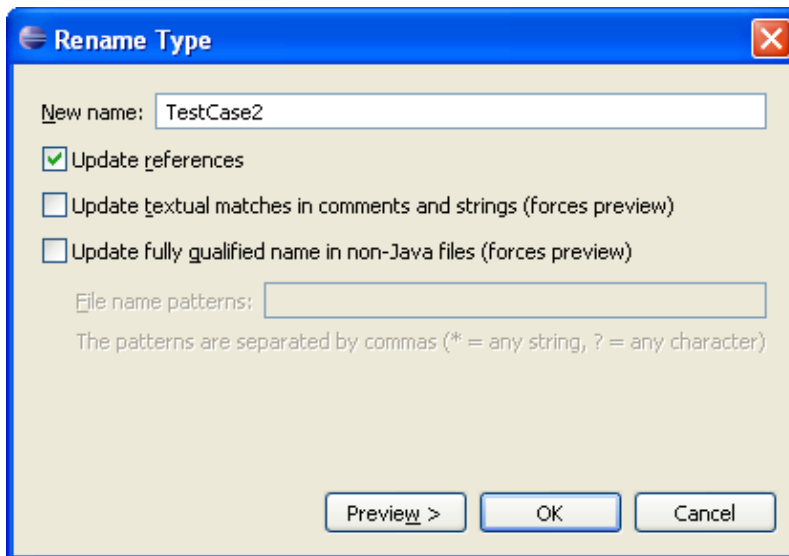
[Code style preference page](#)

[Code templates preference page](#)

Renaming Java elements

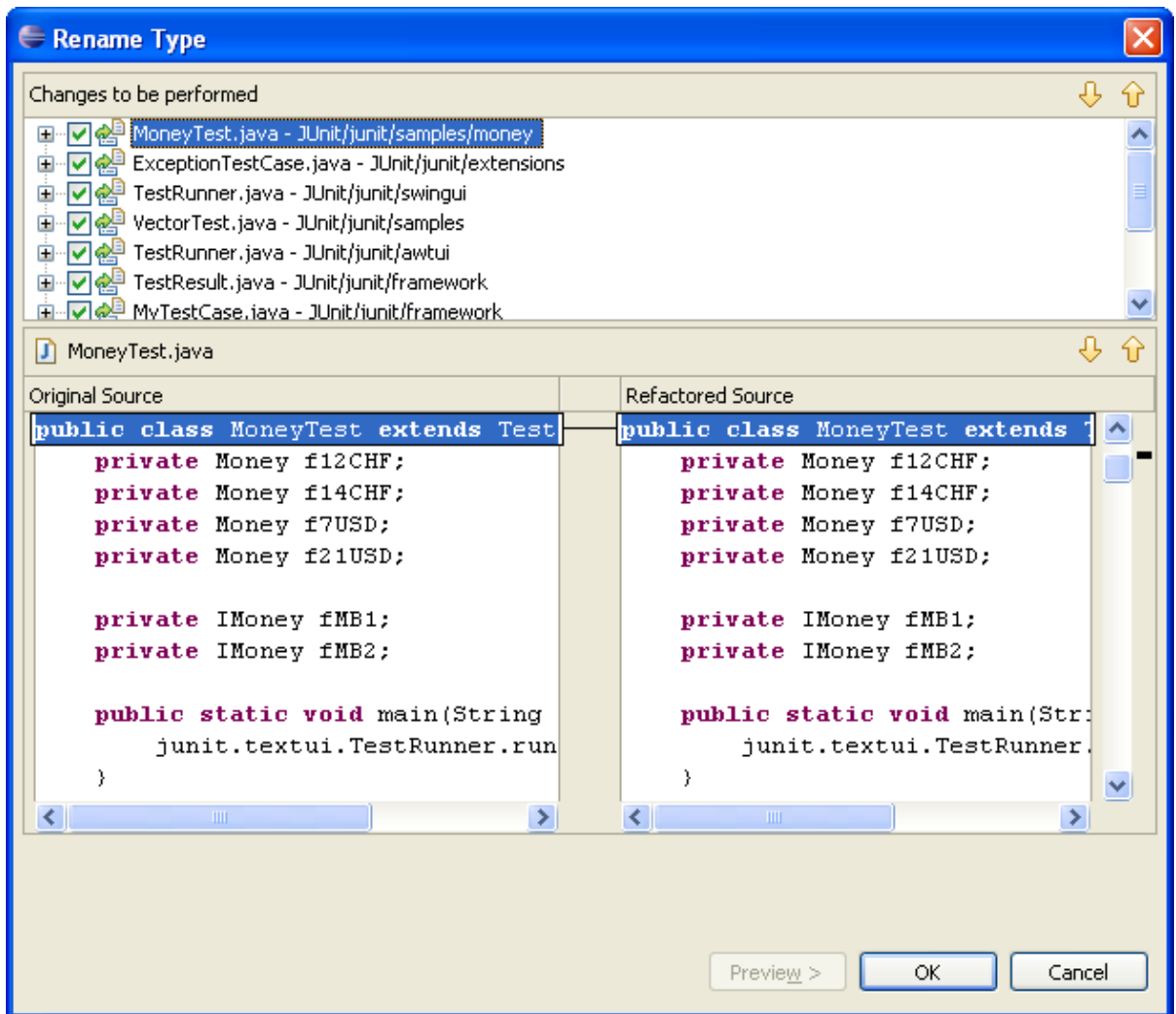
In this section, you will rename a Java element using refactoring. Refactoring actions change the structure of your code without changing its semantic behavior.

1. In the Package Explorer view, select *junit.framework.TestCase.java*.
2. From its context menu, select **Refactor > Rename**.
3. In the **New Name** field on the Rename Compilation Unit page, type "TestCase2".



4. To preview the changes that will be made as a result of renaming the class, press **Preview >**.
5. The workbench analyzes the proposed change and presents you with a preview of the changes that would take place if you rename this resource.

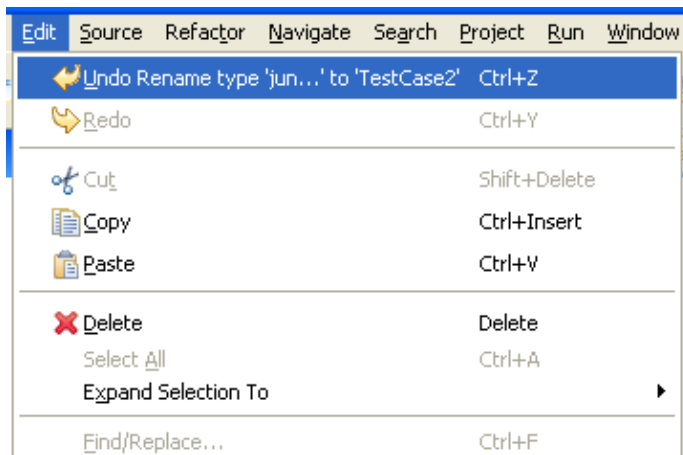
Since renaming a compilation unit will affect the import statements in other compilation units, there are other compilation units affected by the change. These are shown in a list of changes in the preview pane.



6. On the Refactoring preview page, you can scroll through the proposed changes and select or deselect changes, if necessary. You will typically accept all of the proposed changes.
7. Click **OK** to accept all proposed changes.

You have seen that a refactoring action can cause many changes in different compilation units. These changes can be undone as a group.

1. In the menu bar, select **Edit > Undo Rename TestCase.java to TestCase2.java**.



2. The refactoring changes are undone, and the workbench returns to its previous state. You can undo refactoring actions right up until you change and save a compilation unit, at which time the refactoring undo buffer is cleared.

■ Related concepts

Refactoring support

■ Related tasks

Refactoring

Renaming a compilation unit

Refactoring without preview

Refactoring with preview

Previewing refactoring changes

Undoing a refactoring operation

■ Related reference

Refactoring actions

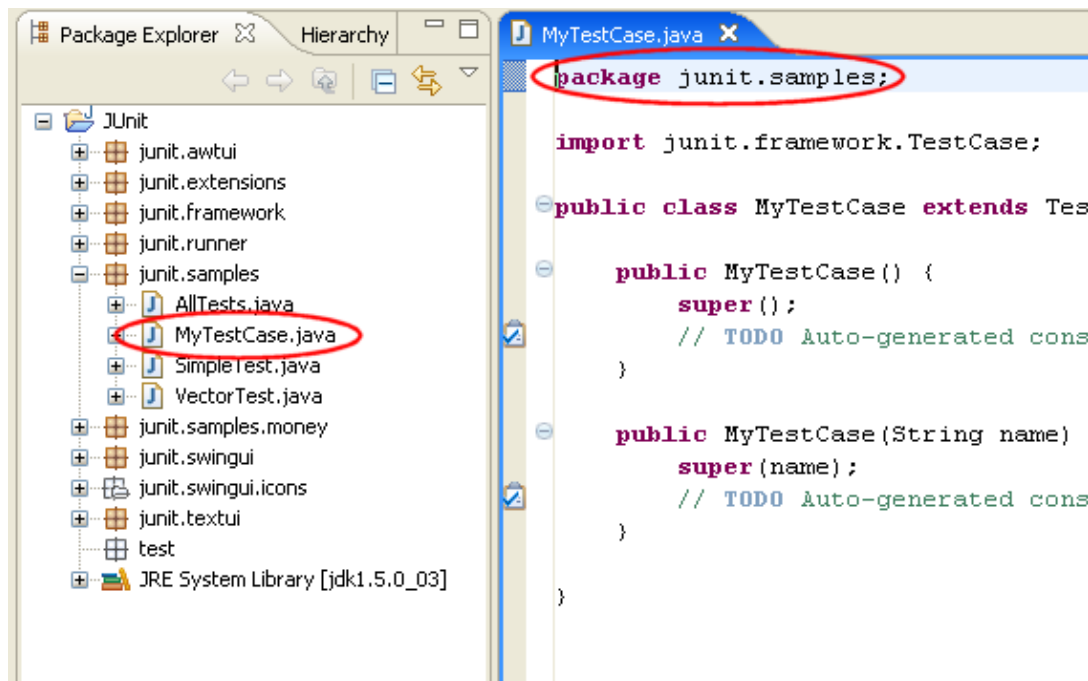
Refactoring wizard

Java preferences

Moving and copying Java elements

In this section, you will use refactoring to move a resource between Java packages. Refactoring actions change the structure of your code without changing its semantic behavior.

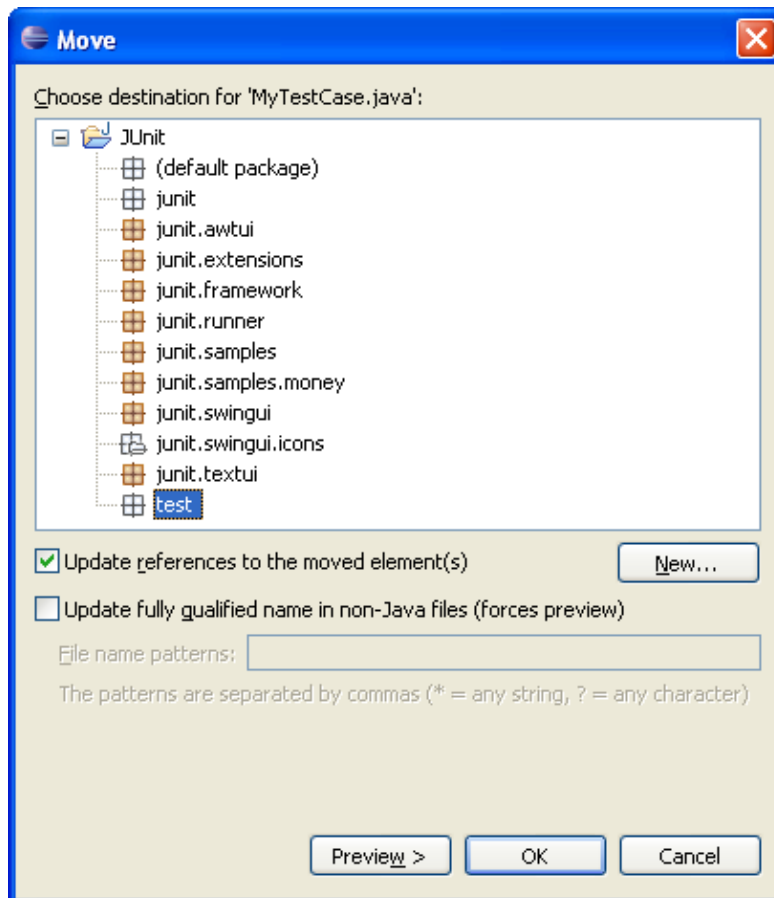
1. In the Package Explorer view, select the *MyTestCase.java* file from the *test* package and drag it into the *junit.samples* package. Dragging and dropping the file is similar to selecting the file and choosing **Refactor > Move** from the context menu.
2. You will be prompted to select whether or not to update references to the file you are moving. Typically, you will want to do this to avoid compile errors. You can press the **Preview** button to see the list of changes that will be made as a result of the move.
3. Press **OK**. The file is moved, and its package declaration changes to reflect the new location.



4. Use **Edit > Undo** to undo the move.

The context menu is an alternative to using drag and drop. When using the menu, you must specify a target package in the Move dialog, in addition to selecting the update references options you've already seen.

1. Select the *MyTestCase.java* file and from its context menu, select **Refactor > Move**.
2. In the Move dialog, expand the hierarchy to browse the possible new locations for the resource. Select the *junit.samples* package, then click **OK**. The class is moved, and its package declaration is updated to the new location.



■ Related concepts

[Java views](#)

[Refactoring support](#)

■ Related tasks

[Refactoring](#)

[Copying and moving Java elements](#)

[Moving folders, packages and files](#)

■ Related reference

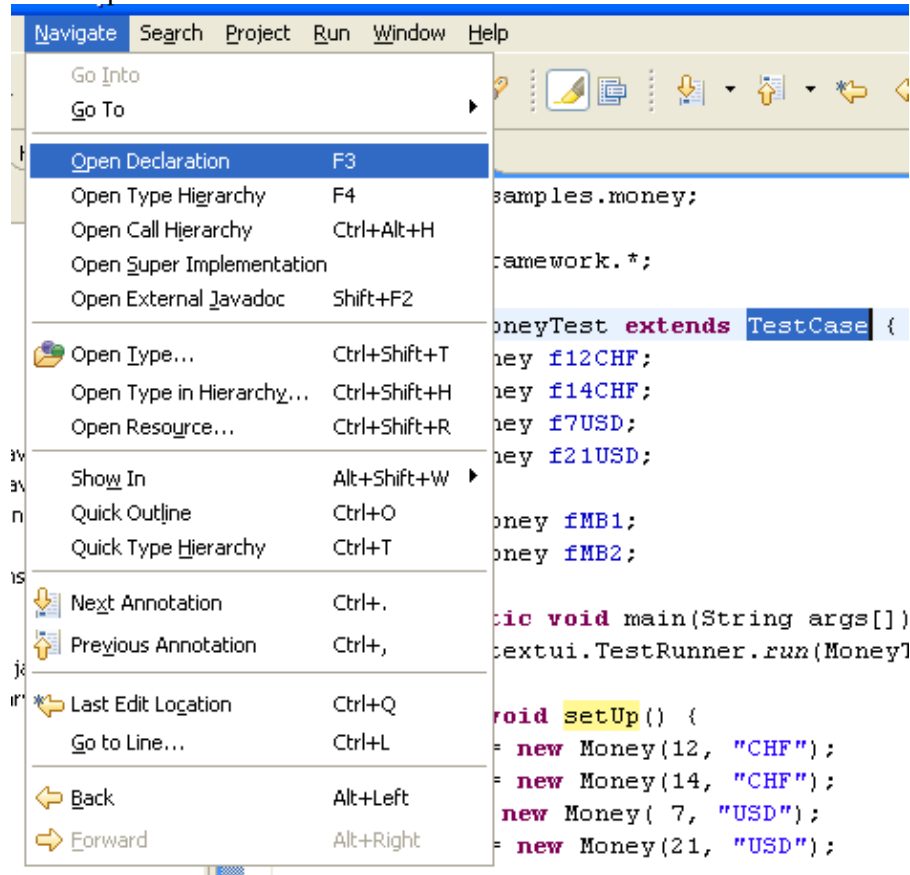
[Refactoring actions](#)

[Refactoring wizard](#)

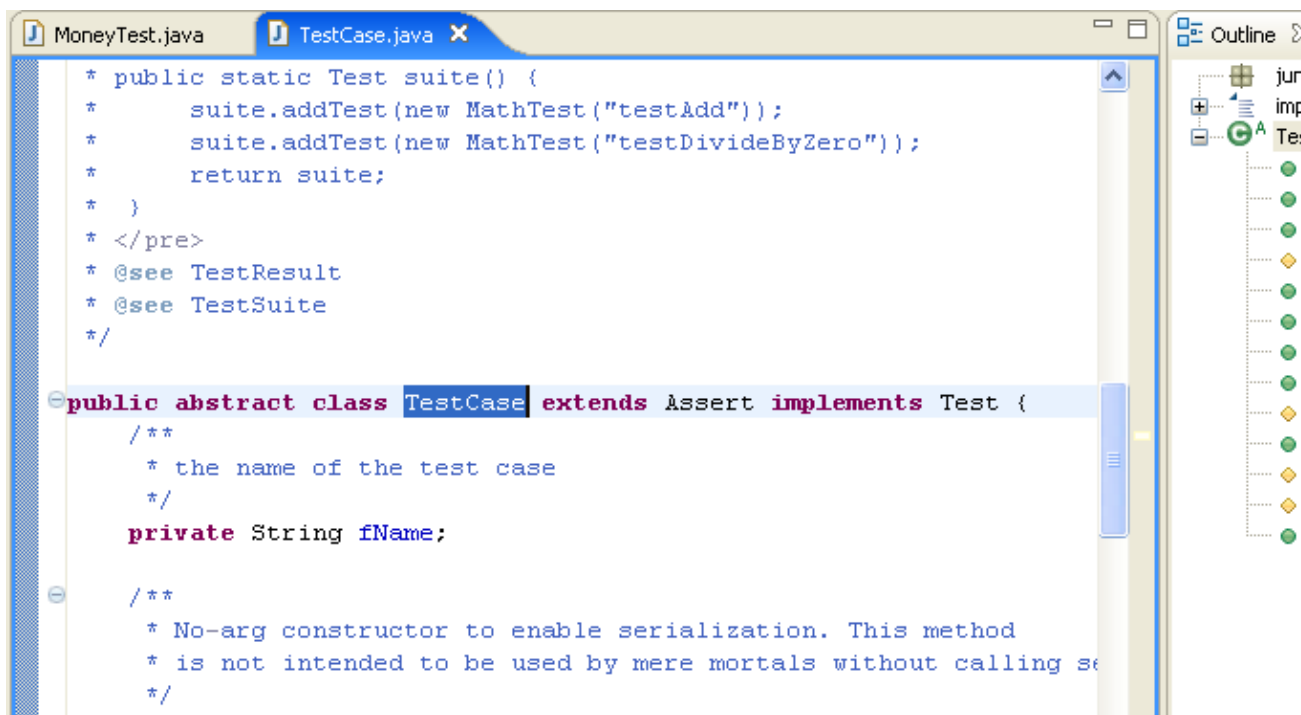
[Java preferences](#)

Navigate to a Java element's declaration

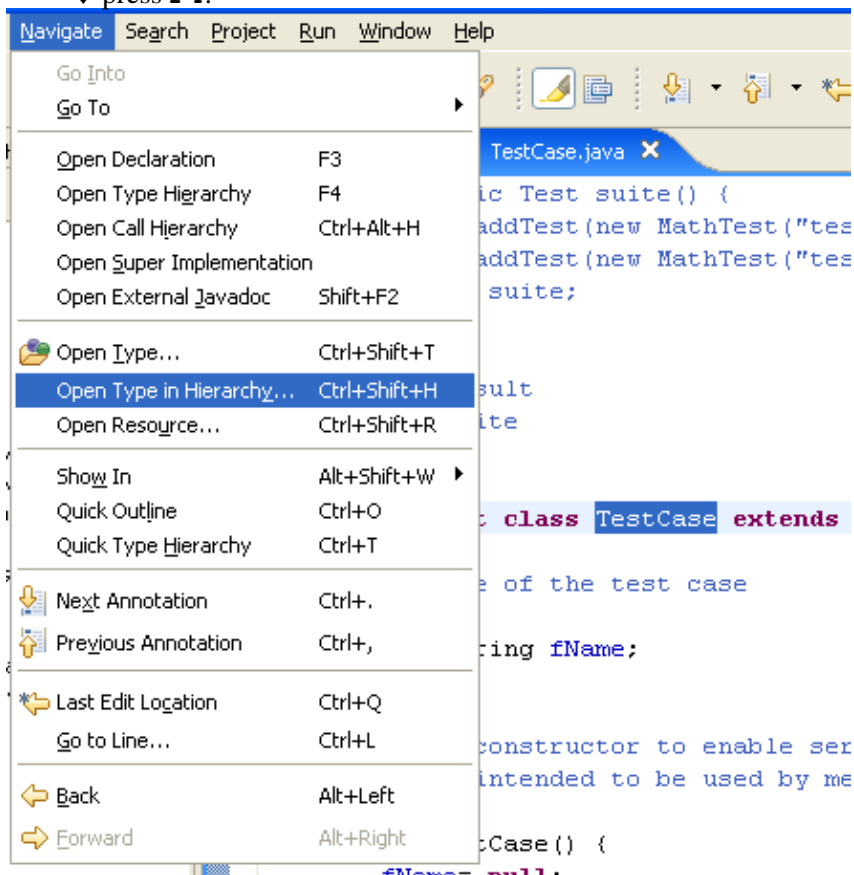
1. Open the `junit.samples.money.MoneyTest.java` file in the Java editor.
2. On the first line of the `MoneyTest` class declaration, select the superclass `TestCase` and either
 - ◆ from the menu bar select `Navigate > Open Declaration` or
 - ◆ press **F3**.



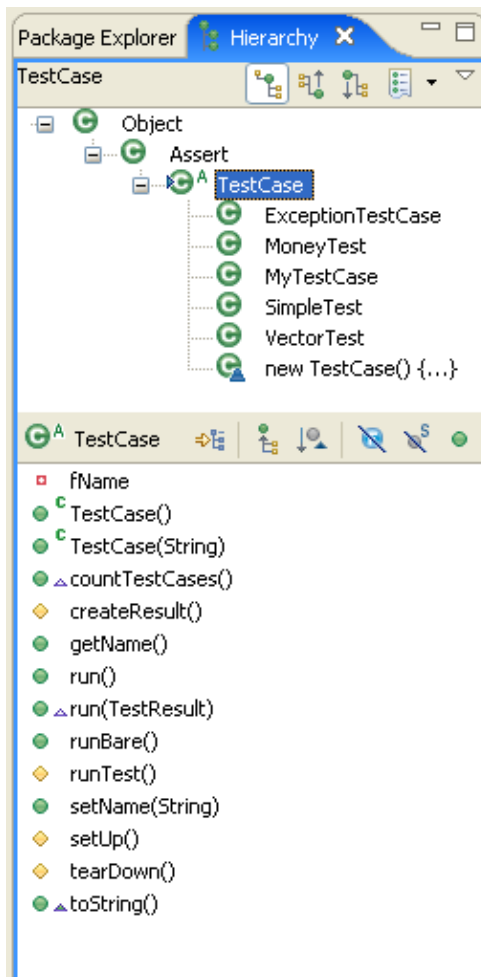
The `TestCase` class opens in the editor area and is also represented in the Outline view.
Note: This command also works on methods and fields.



3. With the *TestCase.java* editor open and the class declaration selected:
 - ◆ from the menu bar select Navigate > Open Type Hierarchy or
 - ◆ press **F4**.



4. The Hierarchy view opens with the *TestCase* class displayed.



Note: You can also open editors on types and methods in the Hierarchy view.

■ Related tasks

[Using the Hierarchy view](#)

[Opening a type hierarchy on a Java element](#)

[Opening a type hierarchy on the current text selection](#)

[Opening an editor for a selected element](#)

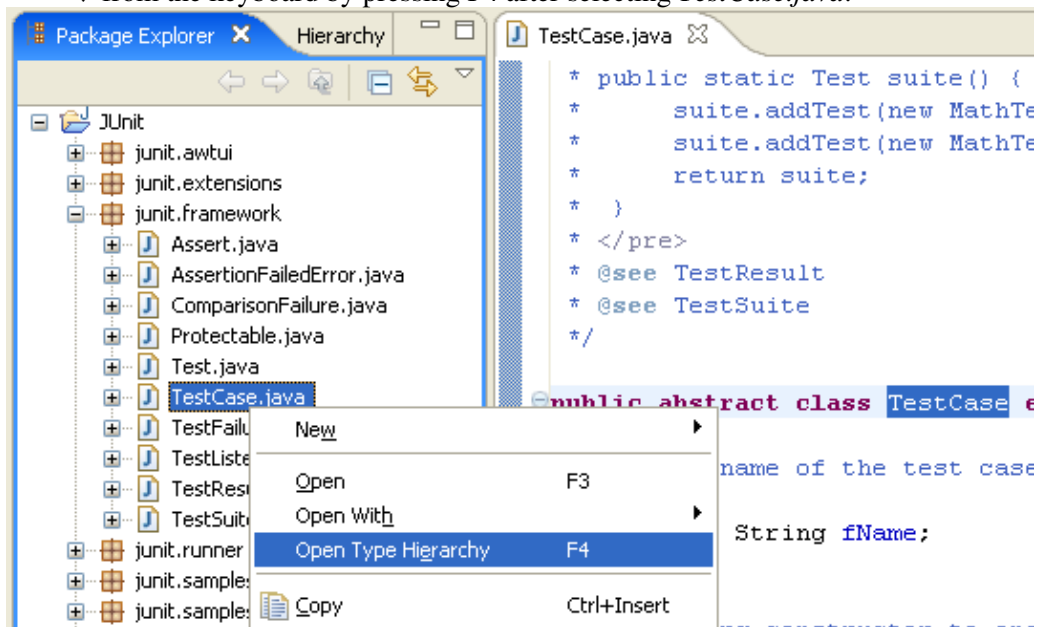
■ Related reference

[Type Hierarchy View](#)

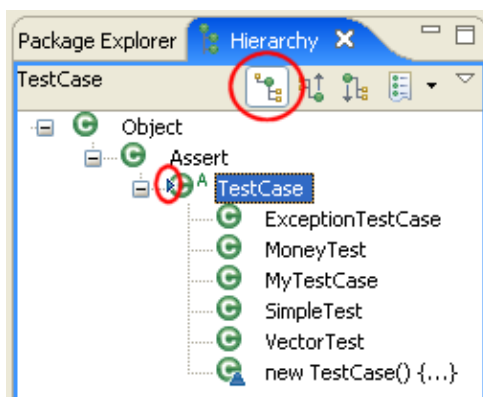
Viewing the type hierarchy

In this section, you will learn about using the Hierarchy view by viewing classes and members in a variety of different ways.

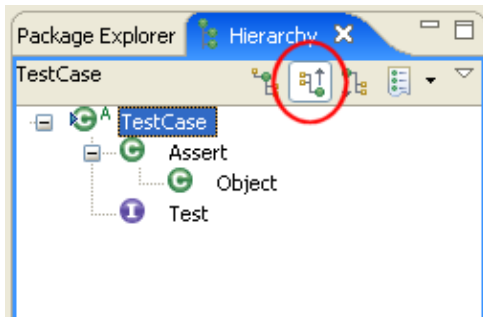
1. In the Package Explorer view, find *junit.framework.TestCase.java*. From its context menu, select **Open Type Hierarchy**. You can also open type hierarchy view:
 - ♦ from the menu bar by selecting **Navigate > Open Type Hierarchy**.
 - ♦ from the keyboard by pressing F4 after selecting *TestCase.java*.



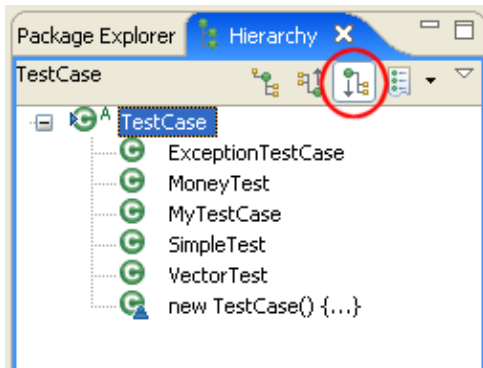
2. The buttons in the view tool bar control which part of the hierarchy is shown. Click the **Show the Type Hierarchy** button to see the class hierarchy, including the base classes and subclasses. The small arrow on the left side of the type icon of *TestCase* indicates that the hierarchy was opened on this type.



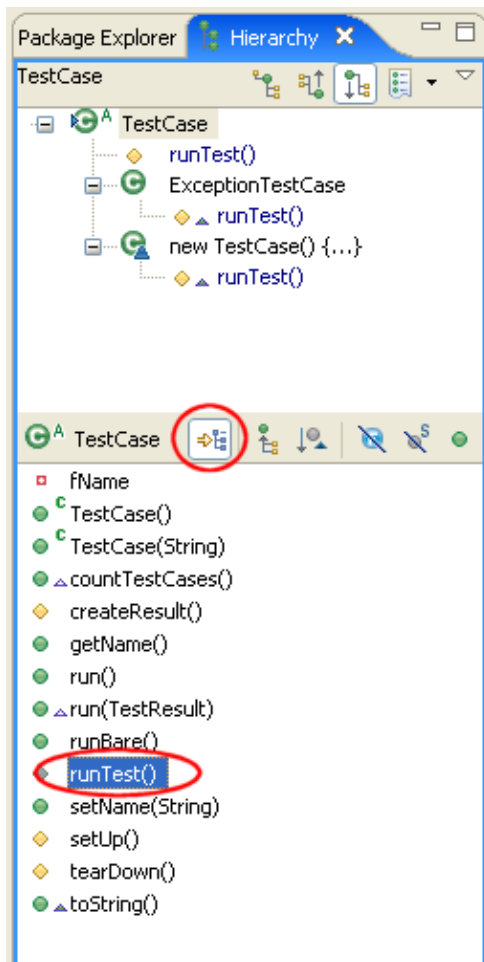
3. Click the **Show the Supertype Hierarchy** button to see a hierarchy showing the type's parent elements including implemented interfaces. This view shows the results of going up the type hierarchy.



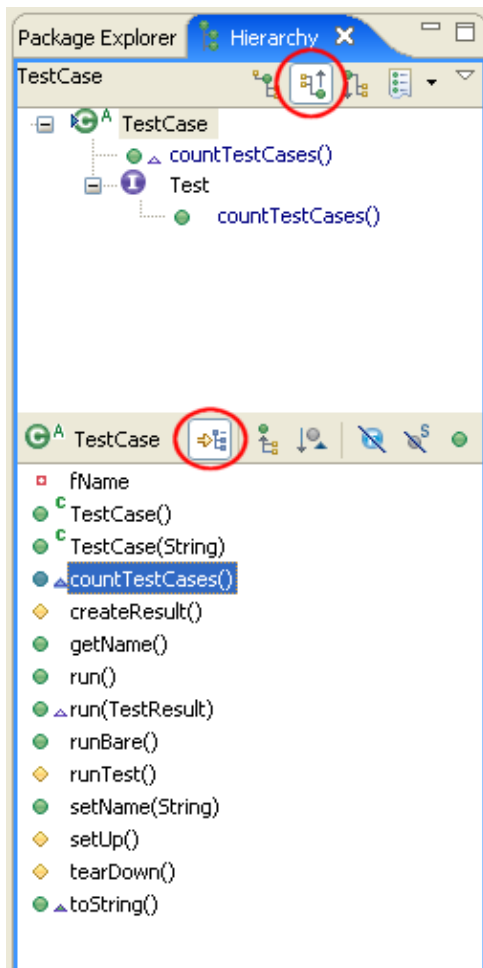
- In this "reversed hierarchy" view, you can see that TestCase implements the Test interface.
4. Click the **Show the Subtype Hierarchy** button in the view toolbar.



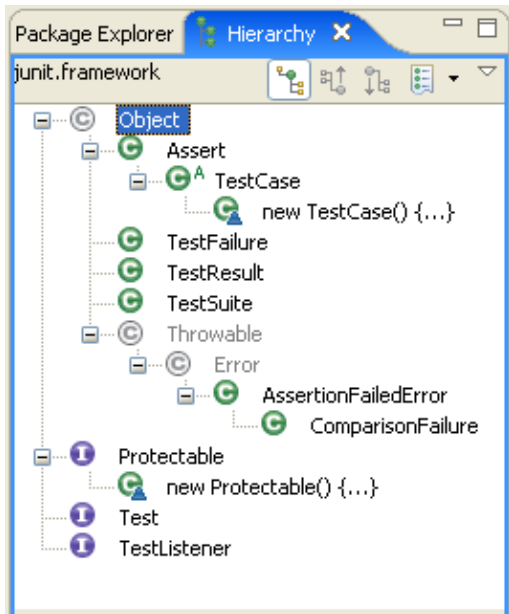
5. Click the **Lock View and Show Members in Hierarchy** button in the toolbar of the member pane, then select the runTest() method in the member pane. The view will now show all the types implementing runTest().



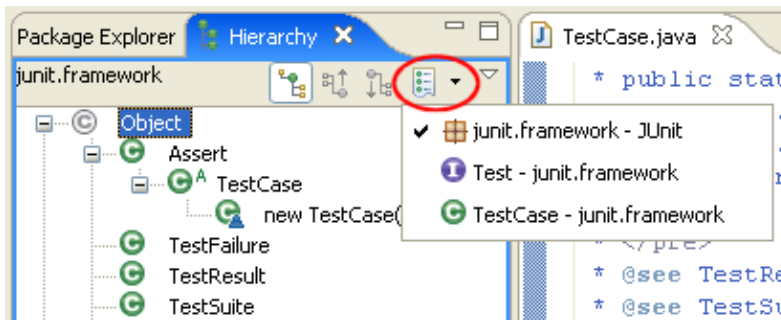
6. In the Hierarchy view, click the **Show the Supertype Hierarchy** button. Then on the member pane, select `countTestCases()` to display the places where this method is declared.



7. In the Hierarchy view select the *Test* element and select **Focus On 'Test'** from its context menu. *Test* is presented in the Hierarchy view.
8. Activate the Package Explorer view and select the package junit.framework. Use **Open Type Hierarchy** from its context menu. A hierarchy is opened containing all classes of the package. For completion of the tree, the hierarchy also shows some classes from other packages. These types are shown by a type icon with a white fill.



9. Use **Previous Type Hierarchies** to go back to a previously opened element. Click on the arrow next to the button to see a list of elements or click on the button to edit the history list.



10. From the menu bar, select **Window > Preferences**. Go to **Java** and select **Open a new Type Hierarchy Perspective**. Then click **OK**.
11. In the Hierarchy view, select the **Test** element again, and activate **Open Type Hierarchy** from the Navigate menu bar. The resource containing the selected type is shown in a new perspective (the Hierarchy perspective), and its source is shown in the Java editor. By setting the preference option for viewing type hierarchy perspectives, you can have more than one type hierarchy in your workbench and switch between them as needed. Close the Hierarchy perspective before proceeding to the next step.

Related concepts

Java views

Related tasks

Using the Hierarchy view

Related reference

[Type Hierarchy view](#)

[Java Base preference page](#)

Searching the workbench

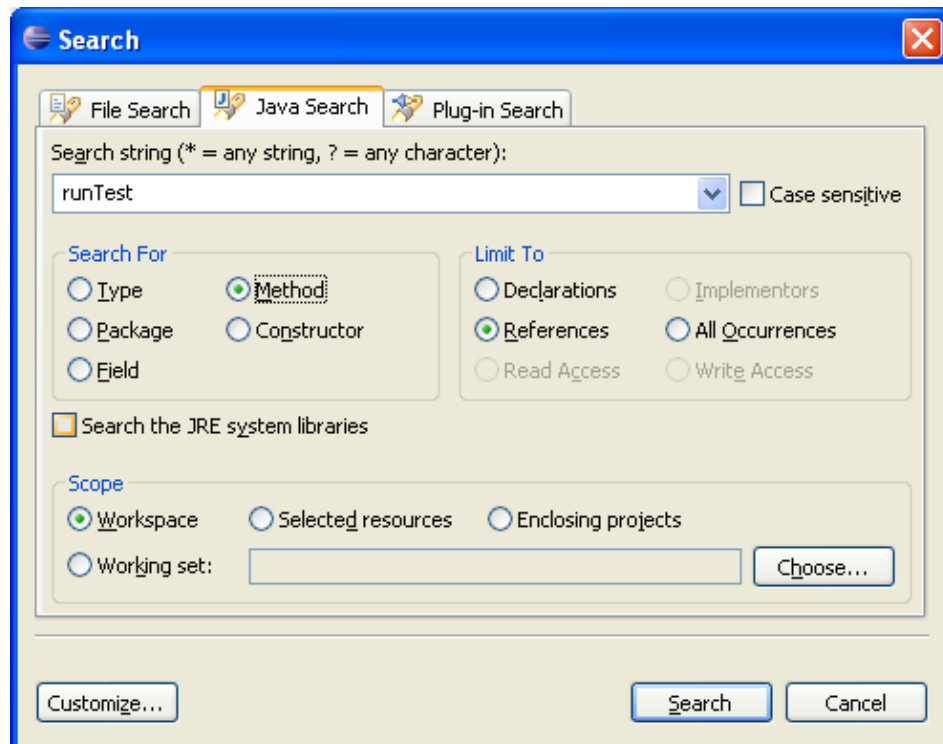
In this section, you will search the workbench for Java elements.

In the Search dialog, you can perform file, text or Java searches. Java searches operate on the structure of the code. File searches operate on the files by name and/or text content. Java searches are faster, since there is an underlying indexing structure for the code structure. Text searches allow you to find matches inside comments and strings.

Performing a Java search from the workbench

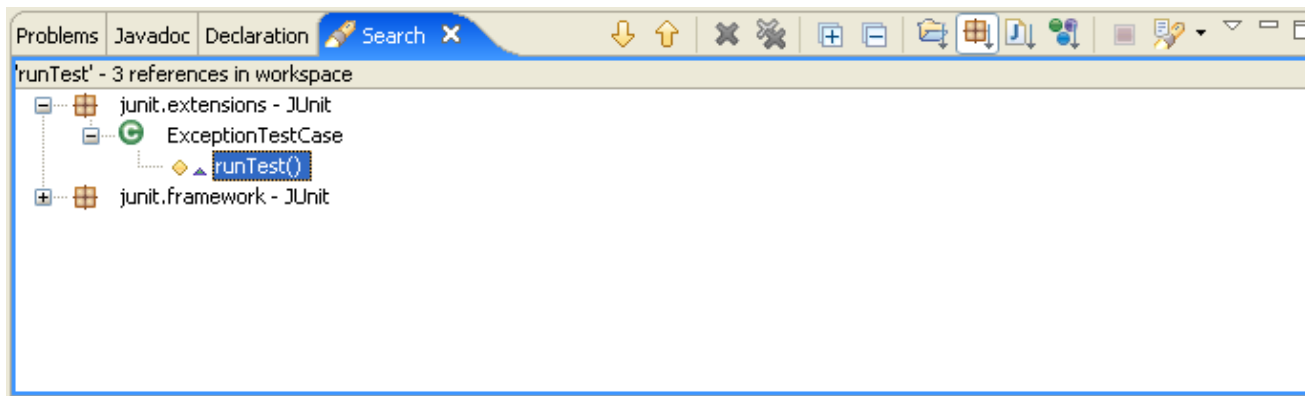
1. In the Java perspective, click the **Search** (🔍) button in the workbench toolbar or use **Search > Java** from the menu bar.
2. If it is not already selected, select the **Java Search** tab.
3. In the **Search string** field, type *runTest*. In the **Search For** area, select **Method**, and in the **Limit To** area, select **References**.

Verify that the Scope is set to **Workspace**.



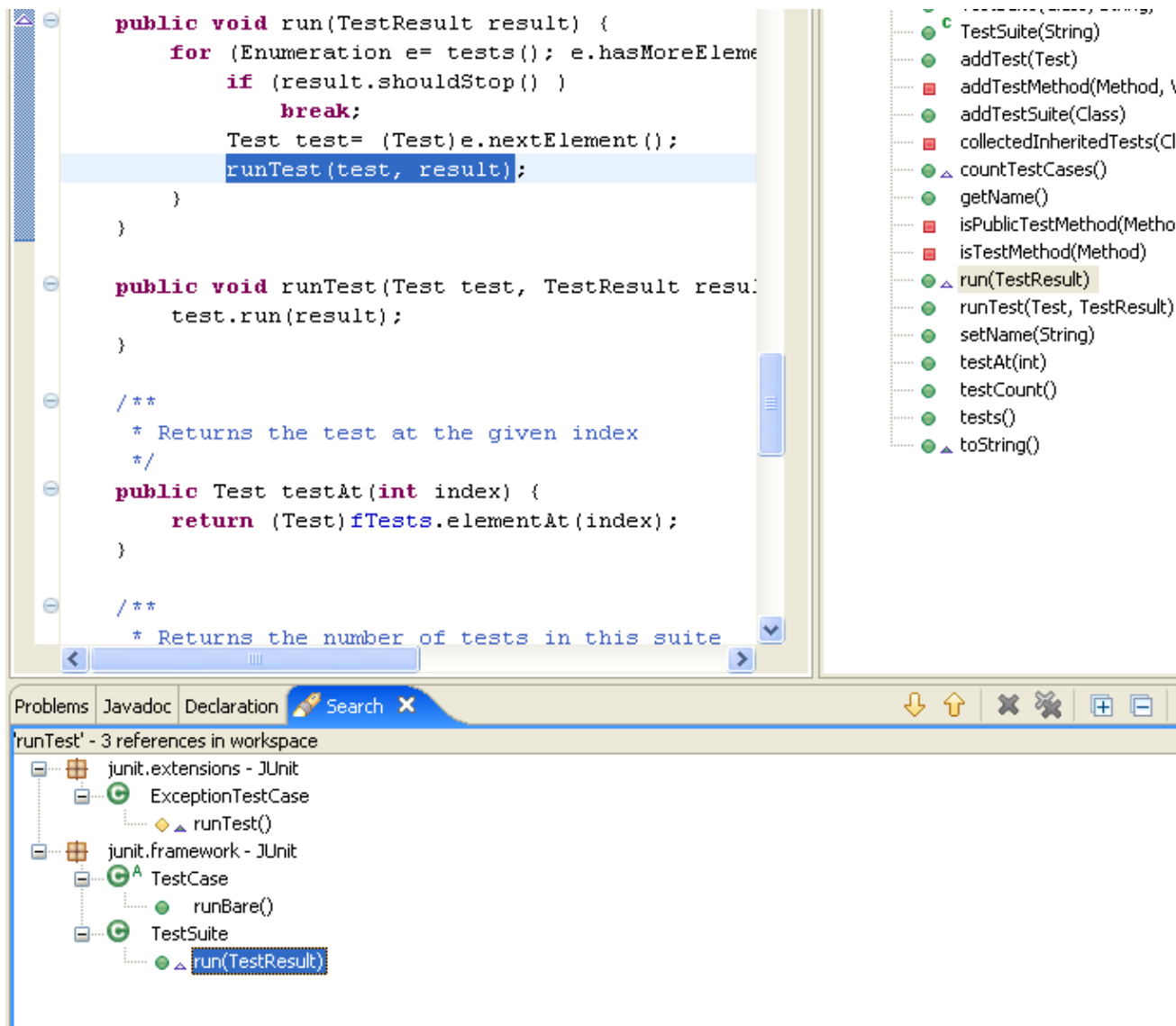
Then click **Search**. While searching you may click **Cancel** at any time to stop the search. Partial results will be shown.

4. In the Java perspective, the Search view shows the search results.



Use the **Show Next Match** (↓) and **Show Previous Match** (↑) buttons to navigate to each match. If the file in which the match was found is not currently open, it is opened in an editor.

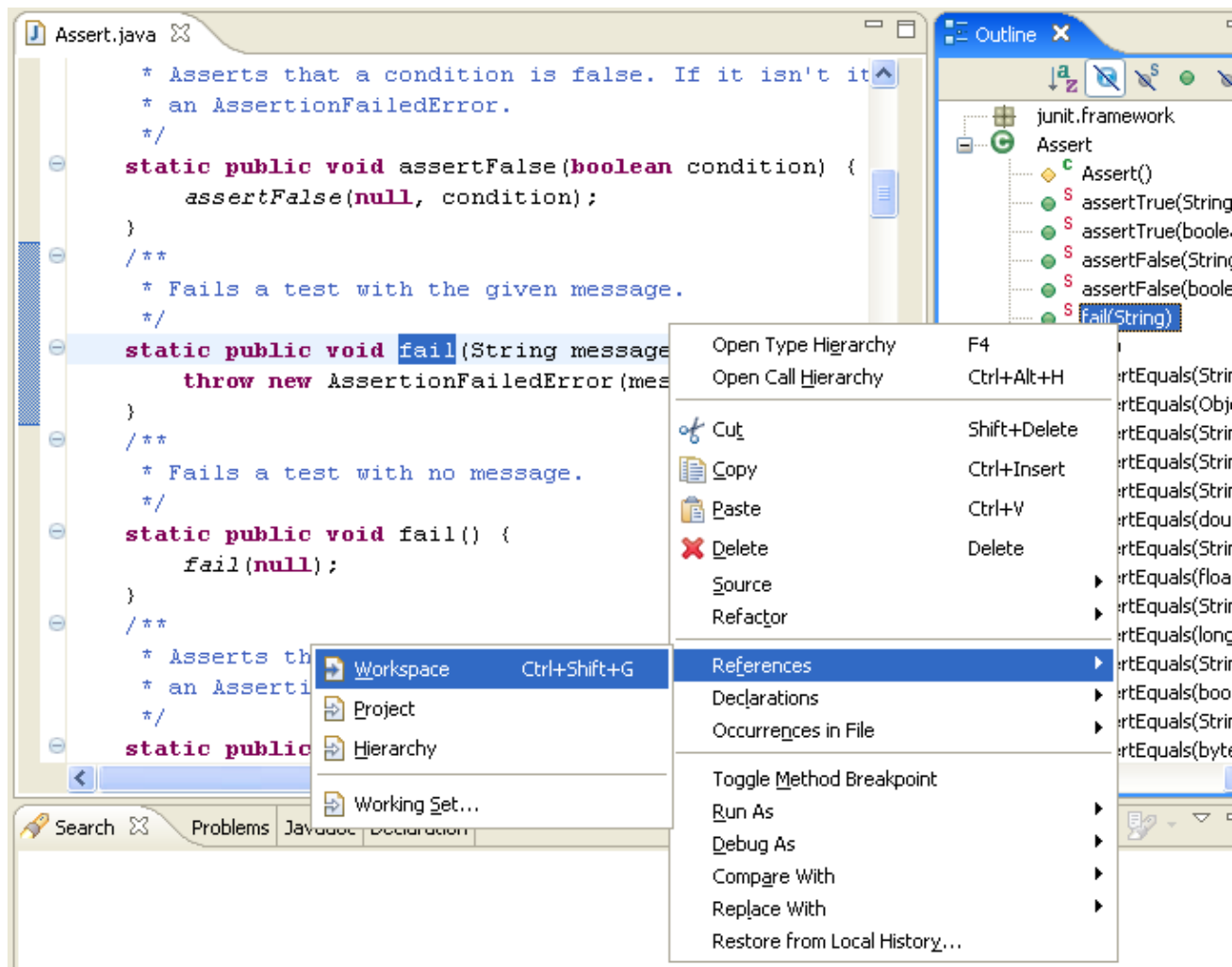
5. When you navigate to a search match using the Search view buttons, the file opens in the editor at the position of the match. Search matches are tagged with a search marker in the vertical ruler.



Searching from a Java view

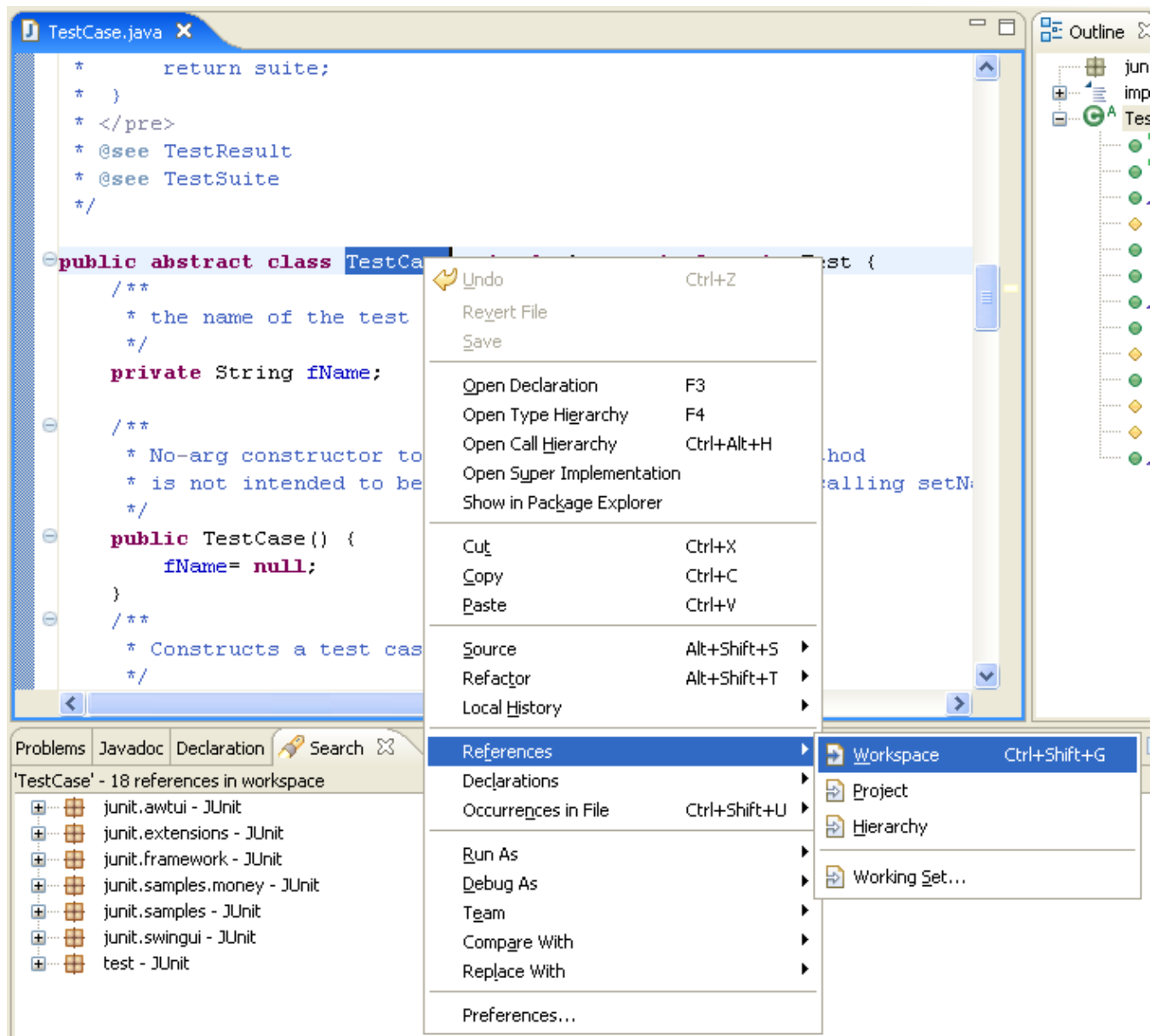
Java searches can also be performed from specific views, including the Outline, Hierarchy view and the Package Explorer view.

1. In the Package Explorer view, double-click *junit.framework.Assert.java* to open it in an editor.
2. In the Outline view, select the `fail(String)` method, and from its context menu, select **References > Workspace**.



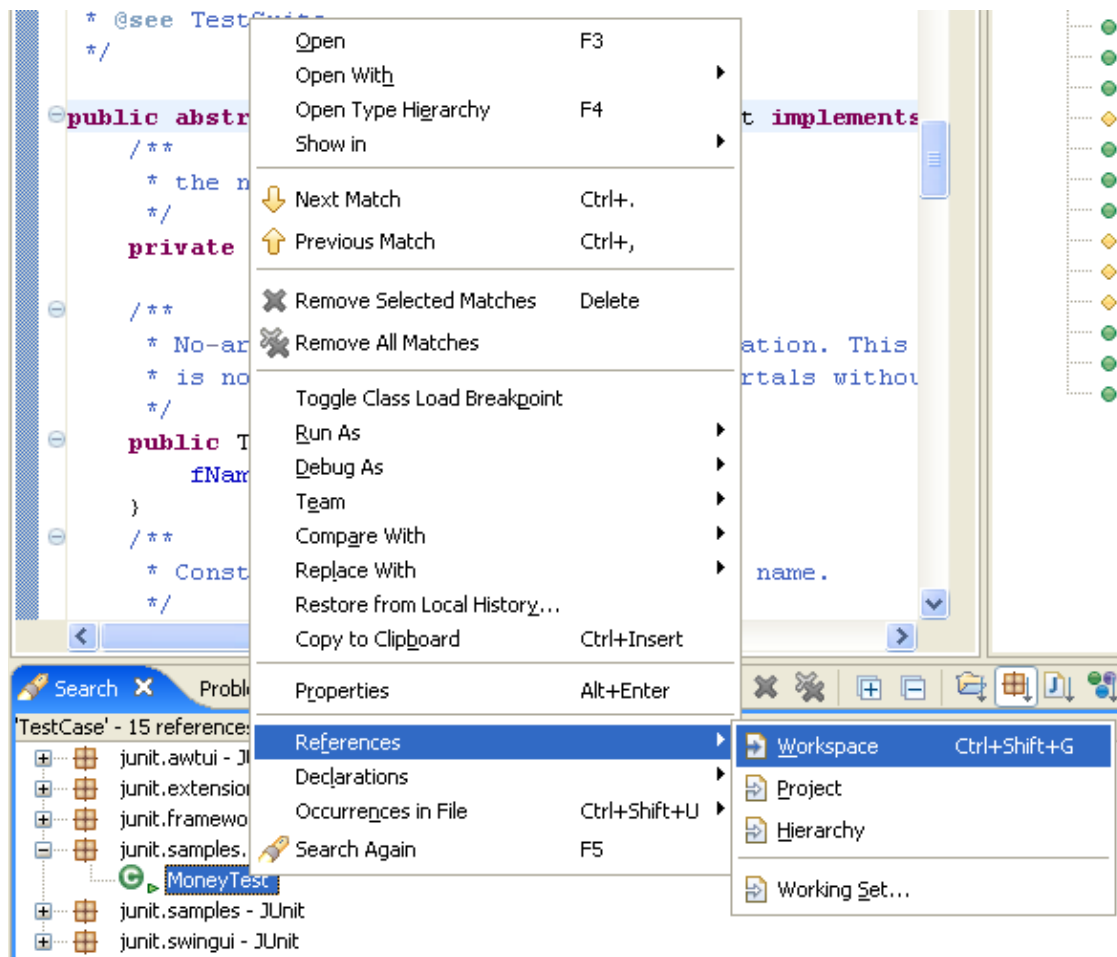
Searching from an editor

From the Package Explorer view, open *junit.framework.TestCase.java*. In the editor, select the class name `TestCase` and from the context menu, select **References > Workspace**.



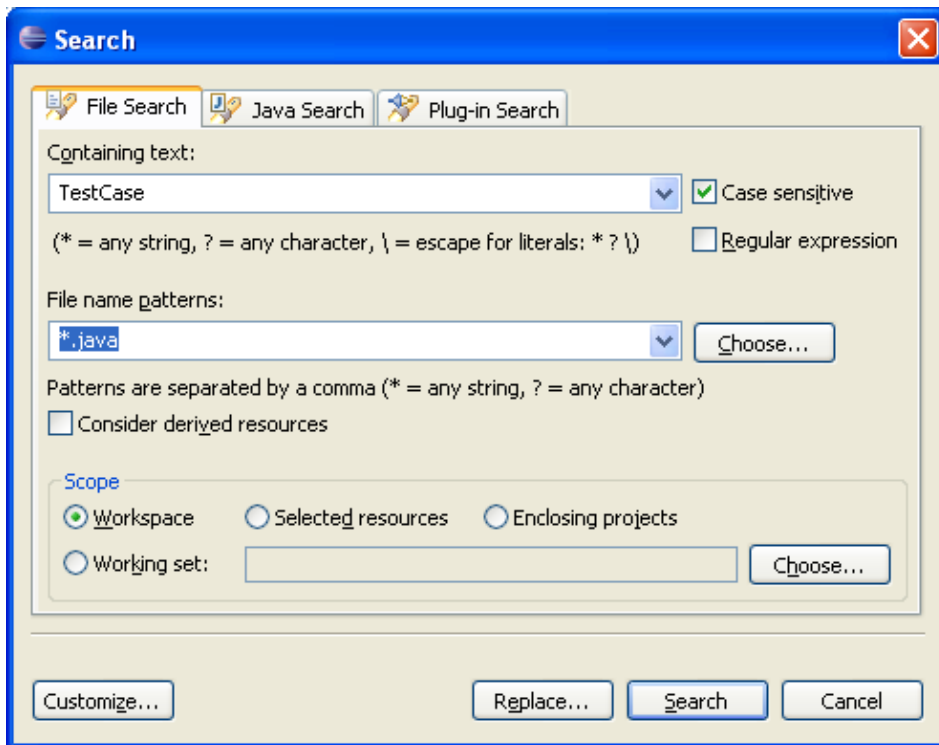
Continuing a search from the search view

The Search Results view shows the results for the TestCase search. Select a search result and open the context menu. You can continue searching the selected element's references and declarations.



Performing a file search

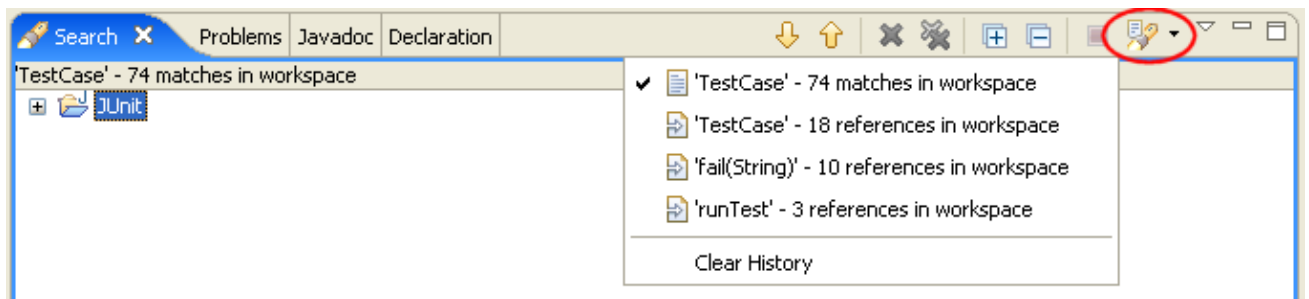
1. In the Java perspective, click the **Search** button in the workbench toolbar or select **Search > File** from the menu bar.
2. If it is not already selected, select the **File Search** tab.
3. In the **Containing text** field, type *TestCase*. Make sure that the **File name patterns** field is set to **.java*. The Scope should be set to *Workspace*. Then click **Search**.



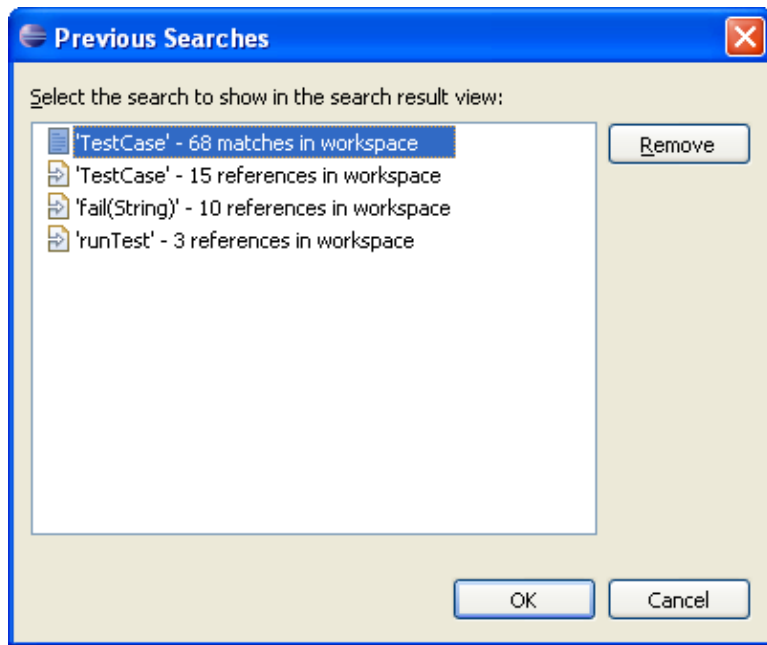
4. To find all files of a given file name pattern, leave the Containing Text field empty.

Viewing previous search results

In the Search Results view, click the arrow next to the **Previous Search Results** toolbar button to see a menu containing the list of the most recent searches. You can choose items from this menu to view previous searches. The list can be cleared by choosing **Clear History**.



The **Previous Search Results** button will display a dialog with the list of all previous searches from the current session.



Selecting a previous search from this dialog will let you view that search.

☒ Related concepts

[Java search](#)

☒ Related tasks

[Conducting a Java search using the search dialog](#)

[Conducting a Java search using pop-up menus](#)

☒ Related reference

[Refactoring actions](#)

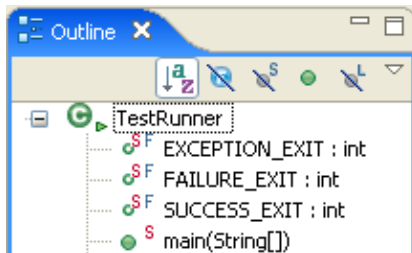
[Refactoring wizard](#)

[Java preferences](#)

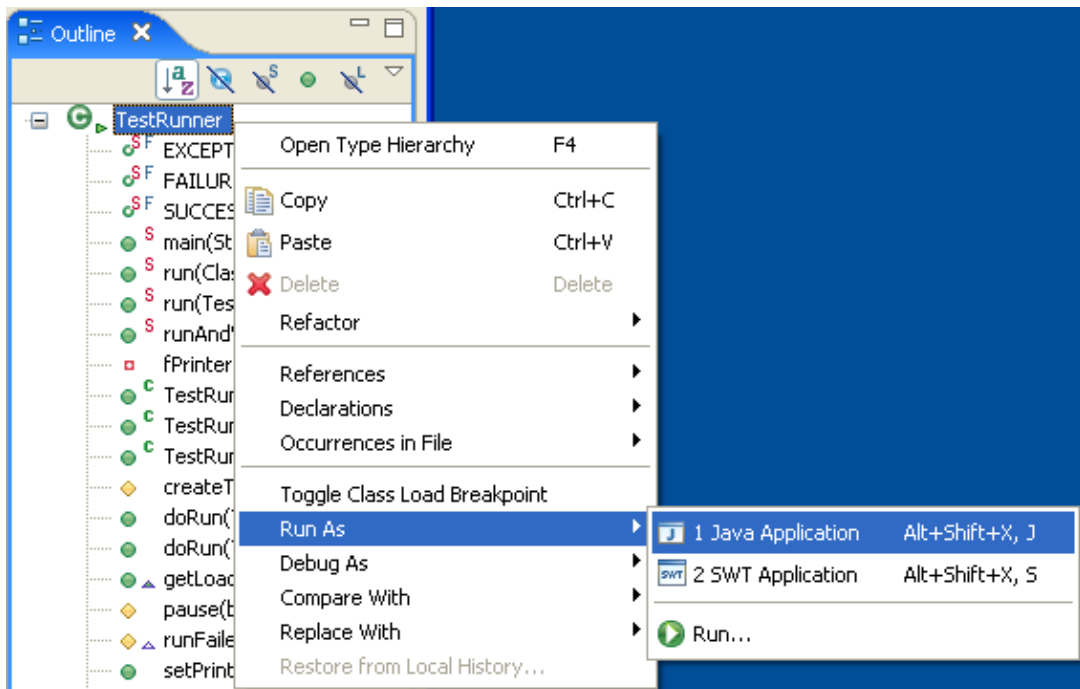
Running your programs

In this section, you will learn more about running Java programs in the workbench.

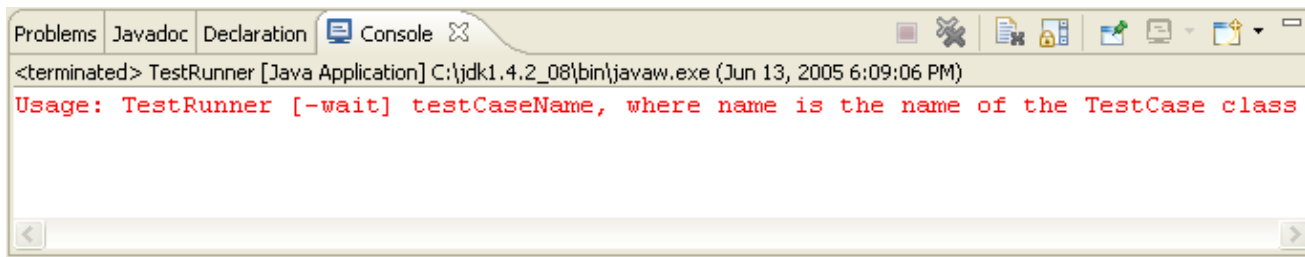
1. In the Package Explorer view, find *junit.textui.TestRunner.java* and double-click it to open it in an editor.
2. In the Outline view, notice that the TestRunner class has an icon which indicates that the class defines a main method.



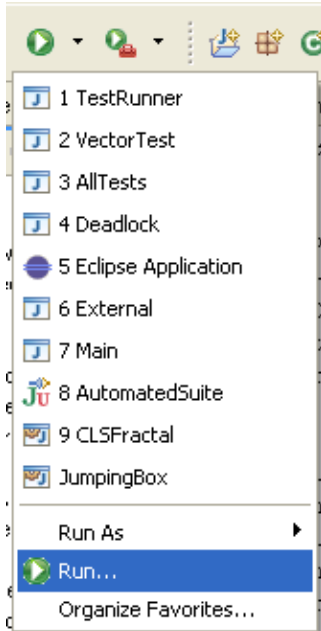
3. Right click on TestRunner.java in the Package Explorer and select **Run As > Java Application**. This will launch the selected class as a local Java application. The **Run As** context menu item is also available in other places, such as the Outline view.



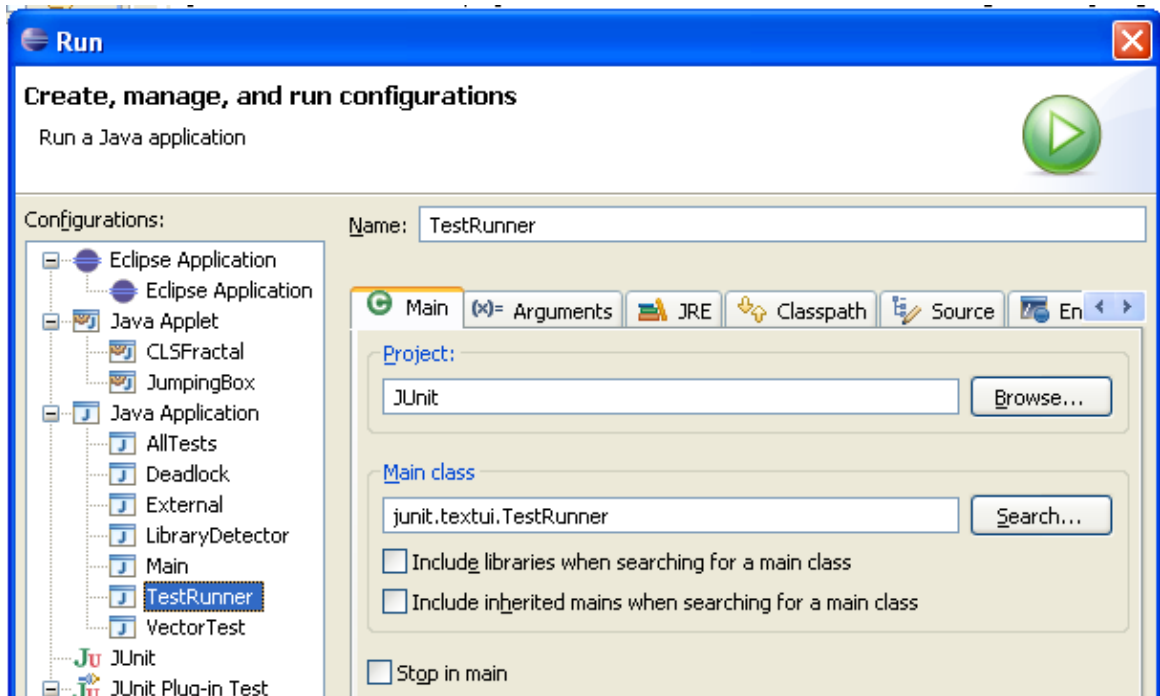
4. Notice that the program has finished running and the following message appears in the Console view telling you that the program needs an execution argument. Running class from the Package Explorer as a Java Application uses the default settings for launching the selected class and does not allow you to specify any arguments.



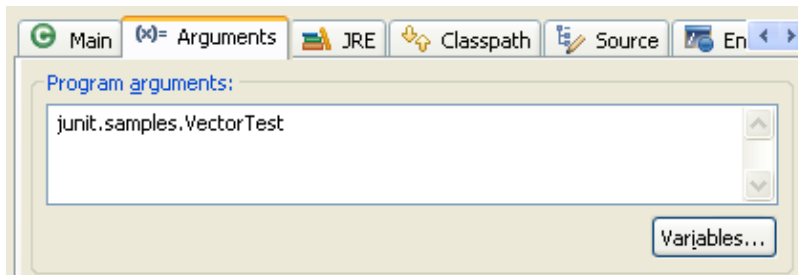
5. To specify arguments, use the drop-down **Run** menu in the toolbar and select **Run....**



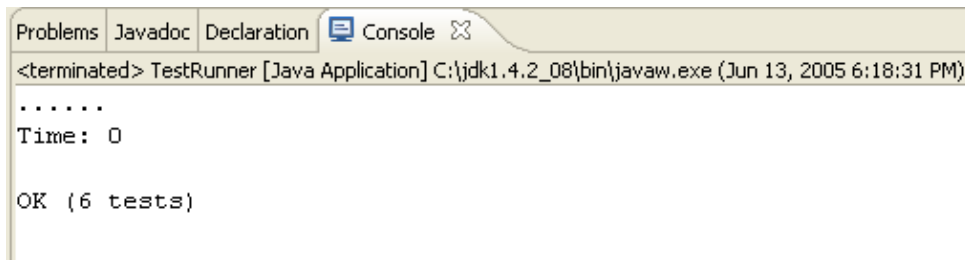
6. This time, the Launch Configurations dialog opens with the TestRunner launch configuration selected. A launch configuration allows you to configure how a program is launched, including its arguments, classpath, and other options. (A default launch configuration was created for you when you chose **Run > Java Application**).



7. Select the Arguments tab and type *junit.samples.VectorTest* in the Program arguments area.



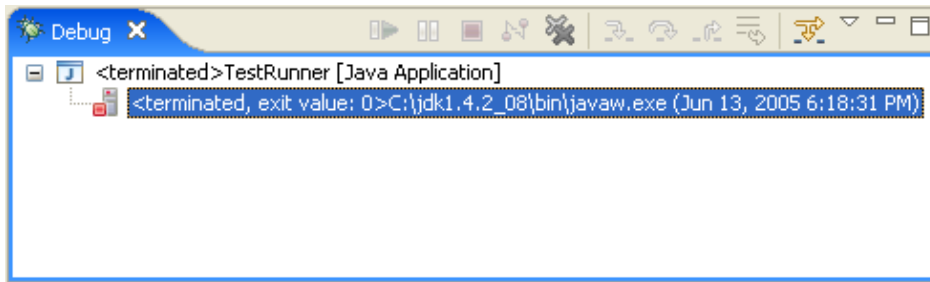
8. Click **Run**. This time the program runs correctly, indicating the number of tests that were run.



9. Switch to the Debug perspective. In the Debug view, notice that a process for the last program launch was registered when the program was run.

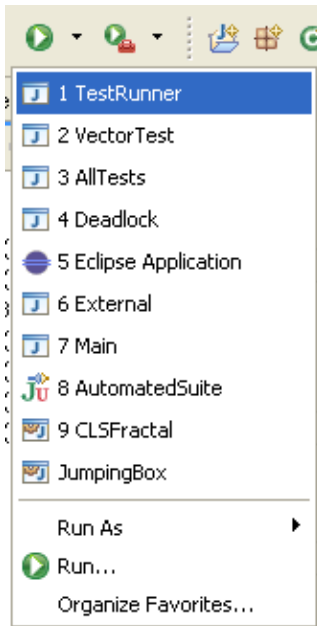
By default, the Debug view automatically removes any terminated launches when a new launch is created. This preference can be configured on the **Launching** preference page located under the **Run/Debug** preference page.

Basic tutorial

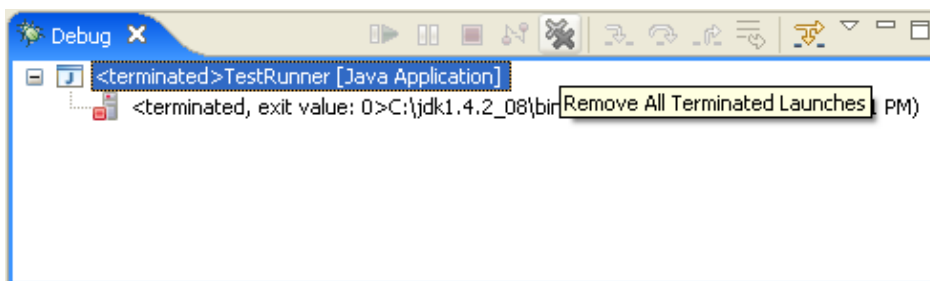


*Note: You can relaunch a terminated process by selecting **Relaunch** from its context menu.*

10. Select the drop-down menu from the **Run** button in the workbench toolbar. This list contains the previously launched programs. These programs can be relaunched by selecting them in the history list.



11. From the context menu in the Debug view (or the equivalent toolbar button), select **Remove All Terminated** to clear the view of terminated launch processes.



Related tasks

[Changing debugger launch options](#)

[Connecting to a remote VM with the Remote Java application launch configuration](#)

[Disconnecting from a VM](#)

[Launching a Java program](#)

[Running and debugging](#)

■ Related reference

[Debug view](#)

[Run and debug actions](#)

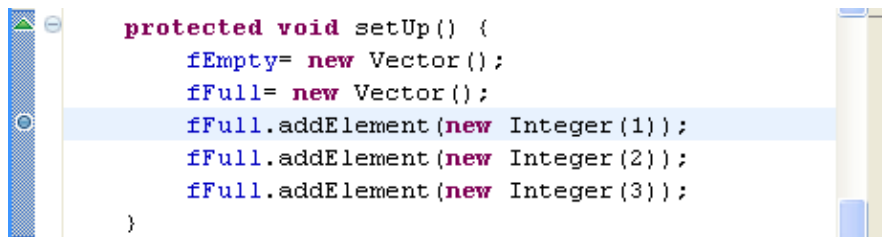
Debugging your programs

In this section, you will debug a Java program.

1. In the Package Explorer view in the Java perspective, double-click *junit.samples.VectorTest.java* to open it in an editor.
2. Place your cursor on the vertical ruler along the left edge of the editor area on the following line in the `setUp()` method:

```
fFull.addElement (new Integer(1));
```

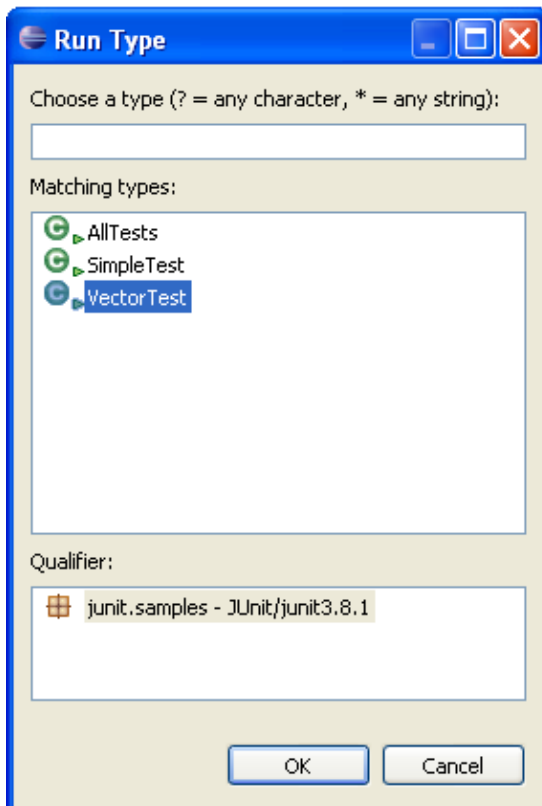
and double-click on the ruler to set a breakpoint.



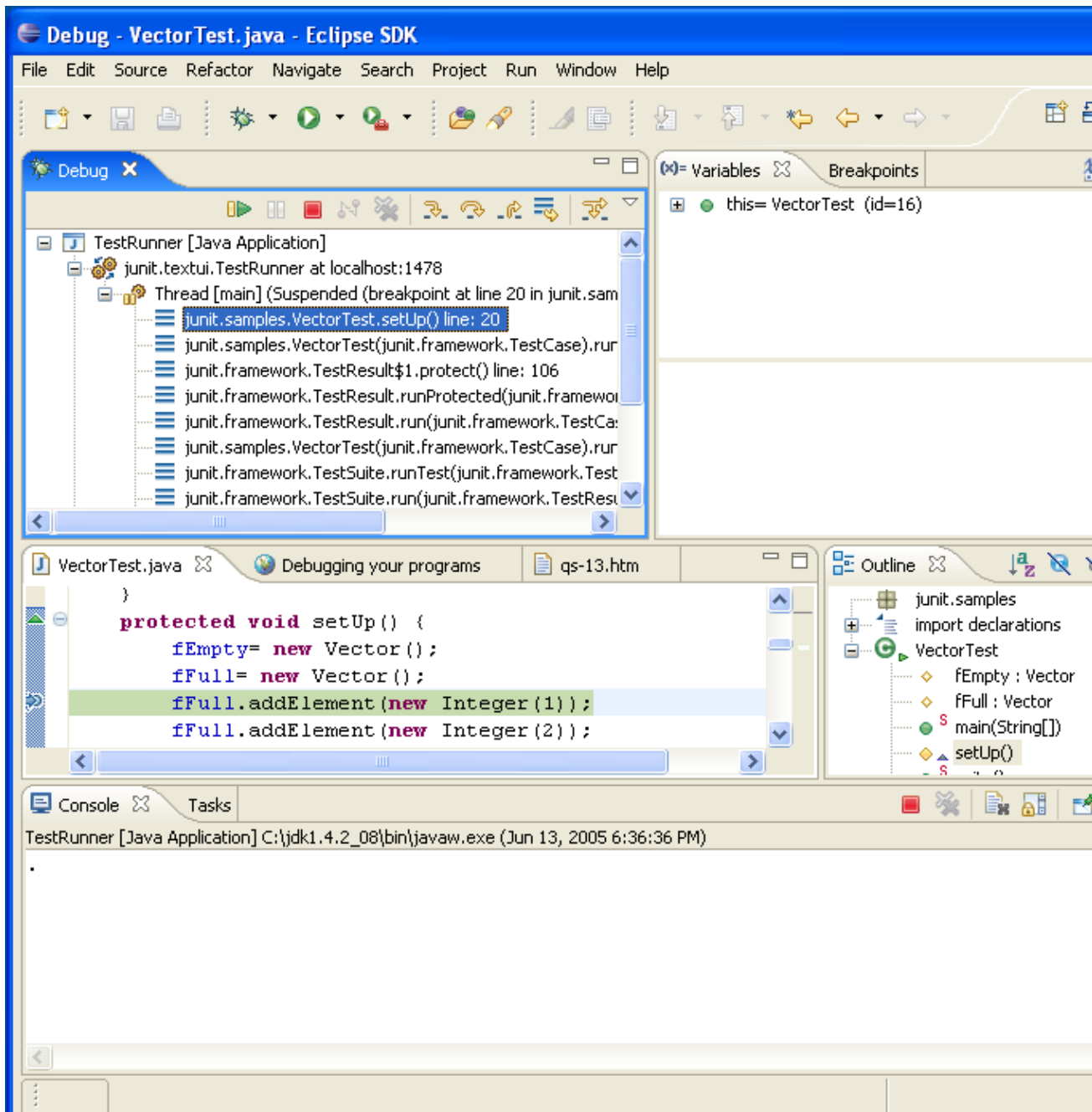
The breakpoint icon indicates the status of the breakpoint. The plain blue breakpoint icon indicates that the breakpoint has been set, but not yet installed.

Note: Once the class is loaded by the Java VM, the breakpoint will be installed and a checkmark overlay will be displayed on the breakpoint icon.

3. In the Package Explorer view, select the *junit.samples* package and select **Debug As**, and then **Java Application**. When you run a program from a package, you will be prompted to choose a type from all classes in the package that define a `main` method.
4. Select the *VectorTest* item in the dialog, then click **OK**.



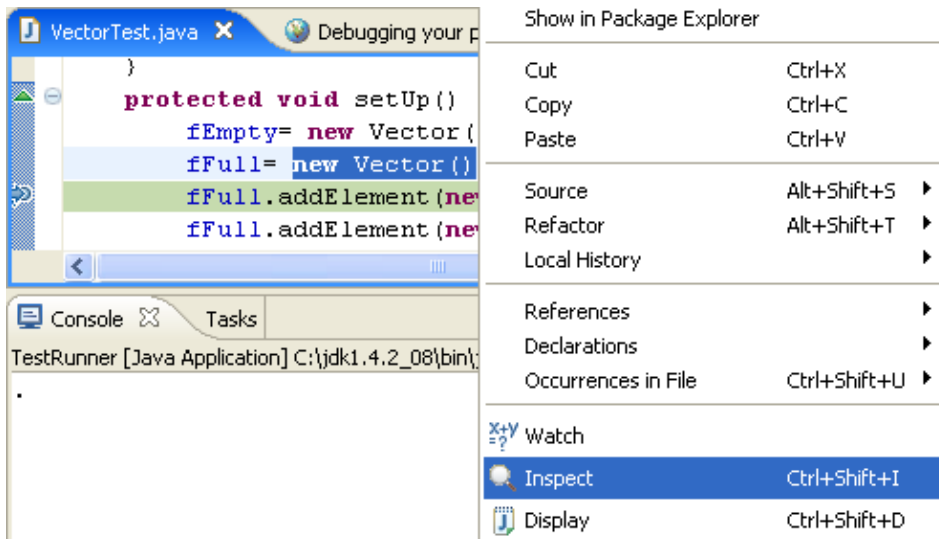
5. The program will run until the breakpoint is reached. When the breakpoint is hit, execution is suspended, and you are asked whether to open the Debug perspective. Click *Yes*. Notice that the process is still active (not terminated) in the Debug view. Other threads might still be running.



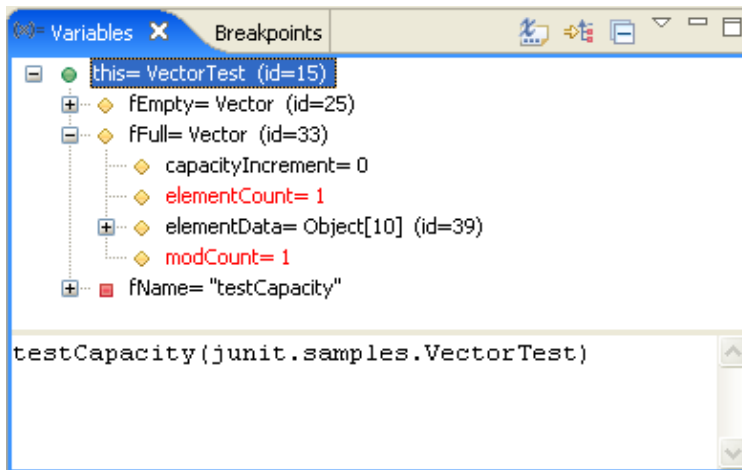
Note: The breakpoint now has a checkmark overlay since the class VectorTest was loaded in the Java VM.

6. In the editor in the Debug perspective, select `new Vector()` from the line above where the breakpoint is set, and from its context menu, select **Inspect**.

Basic tutorial



- The expression is evaluated in the context of the current stack frame, and a pop-up appears which displays the results. You can send a result to the Expressions view by pressing the key binding displayed in the pop-up.
- Expressions that you evaluate while debugging a program will be listed in this view. To delete an expression after working with it, select the expression and choose **Remove** from its context menu.
- The Variables view (available on a tab along with the Expressions view) displays the values of the variables in the selected stack frame. Expand the `this.fFull` tree in the Variables view until you can see `elementCount`.
- The variables (e.g., `elementCount`) in the Variables view will change when you step through `VectorTest` in the Debug view. To step through the code, click the **Step Over** (🔍) button. Execution will continue at the next line in the same method (or, if you are at the end of a method, it will continue in the method from which the current method was called).



- Try some other step buttons (**Step Into** (🔍), **Step Return** (🔍)) to step through the code. Note the differences in stepping techniques.
- You can end a debugging session by allowing the program to run to completion or by terminating it.
 - You can continue to step over the code with the **Step** buttons until the program completes.
 - You can click the **Resume** (▶) button to allow the program to run until the next breakpoint is encountered or until the program is completed.
 - You can select **Terminate** from the context menu of the program's process in the Debug view to terminate the program.

■ Related concepts

[Breakpoints](#)

[Remote debugging](#)

[Local debugging](#)

■ Related tasks

[Adding breakpoints](#)

[Resuming the execution of suspended threads](#)

[Running and debugging](#)

[Suspending threads](#)

■ Related reference

[Debug preferences](#)

[Debug view](#)

[Run and debug actions](#)

[Breakpoints view](#)

[Console view](#)

[Display view](#)

[Expressions view](#)

[Variables view](#)

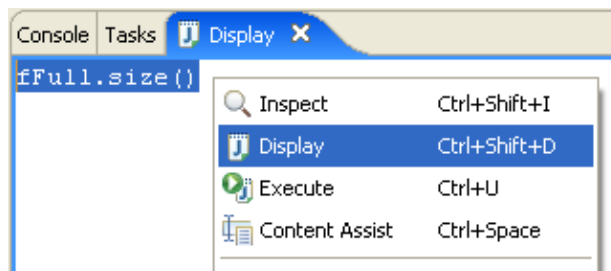
Evaluating expressions

In this section, you will evaluate expressions in the context of your running Java program.

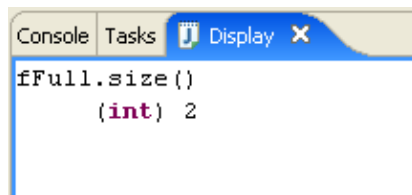
1. Debug `junit.samples.VectorTest.java` to the breakpoint in the `setUp()` method and select **Step Over** twice to populate `fFull`. (See the [Debugging your Programs](#) section for full details.)
2. Open the Display view by selecting **Window > Show View > Display** and type the following line in the view:

```
fFull.size()
```

3. Select the text you just typed, and from its context menu, select **Display**. (You can also choose **Display Result of Evaluating Selected Text** (🔍) from the Display view toolbar.)



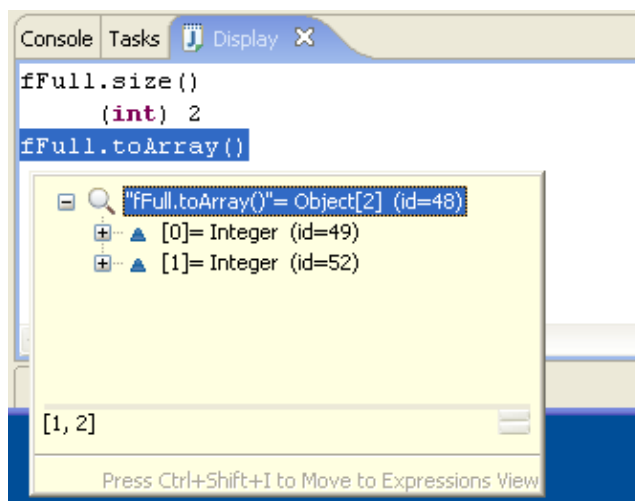
4. The expression is evaluated and the result is displayed in the Display view.



5. On a new line in the Display view, type the following line:

```
fFull.toArray()
```

6. Select this line, and select **Inspect** from the context menu. (You can also choose **Inspect Result of Evaluating Selected Text** (🔍) from the Display view toolbar.)
7. A lightweight window opens with the value of the evaluated expression.



■ Related concepts

Debugger

■ Related tasks

Evaluating expressions

Displaying the result of evaluating an expression

Inspecting the result of evaluating an expression

Viewing compilation errors and warnings

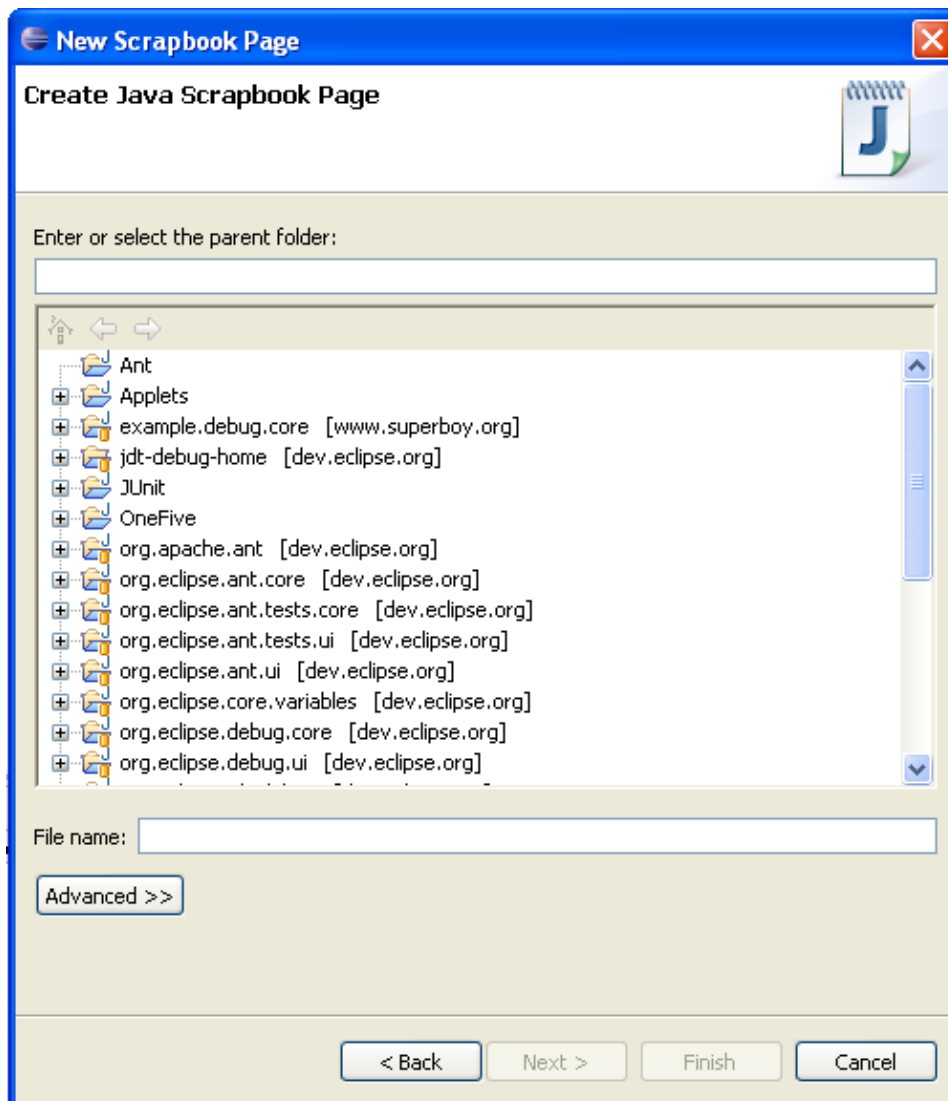
■ Related reference

Expressions view

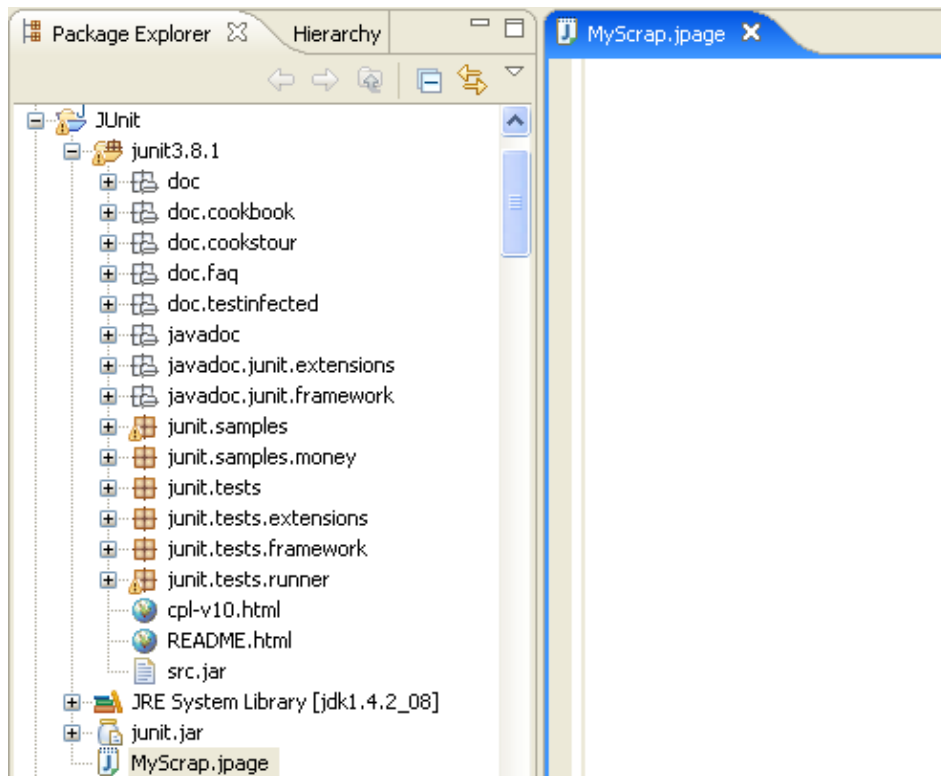
Evaluating snippets

In this section, you will evaluate Java expressions using the Java scrapbook. Java scrapbook pages allow you to experiment with Java code fragments before putting them in your program.

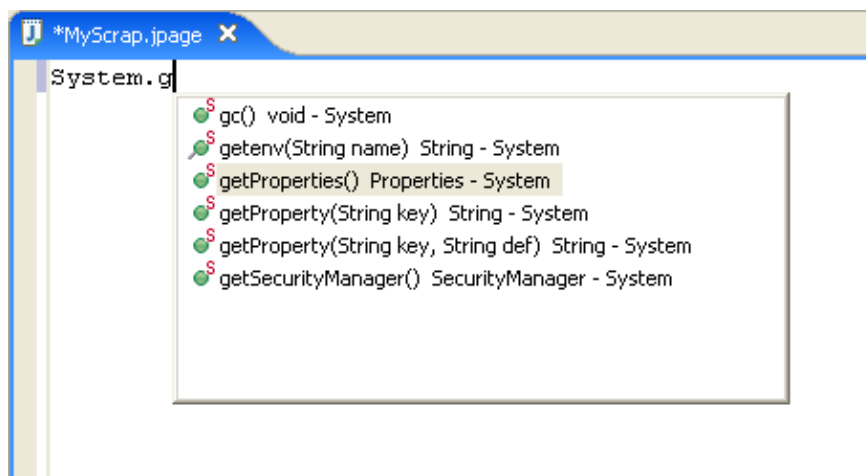
1. In the File menu select **New > Other > Java > Java Run/Debug > Scrapbook Page**. You will be prompted for a folder destination for the page.



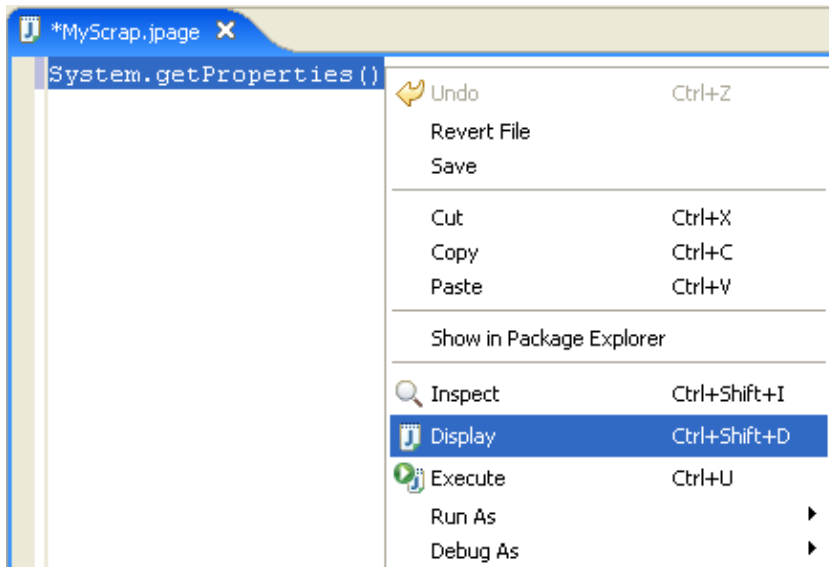
2. In the **Enter or select the folder** field, type or browse below to select the *JUnit* project root directory.
3. In the **File name** field, type *MyScrap*.
4. Click **Finish** when you are done. A scrapbook page resource is created for you with the *jpage* file extension. (The *jpage* file extension will be added automatically if you do not enter it yourself.) The scrapbook page opens automatically in an editor.



5. In the editor, type `System.get` and then use content assist (`Ctrl+Space`) to complete the snippet as `System.getProperties()`.



6. Select the entire line you just typed and select **Display** from the context menu. You can also select **Display Result of Evaluating Selected Text** from the toolbar.



7. When the evaluation completes, the result of the evaluation is displayed and highlighted in the scrapbook page.
8. You can inspect the result of an evaluation by selecting text and choosing ***Inspect*** from the context menu (or selecting ***Inspect Result of Evaluating Selected Text*** from the toolbar.)
9. When you are finished evaluating code snippets in the scrapbook page, you can close the editor. Save the changes in the page if you want to keep the snippets for future use.

■ Related concepts

[Debugger](#)

■ Related tasks

[Creating a Java scrapbook page](#)

[Displaying the result of evaluating an expression](#)

[Inspecting the result of evaluating an expression](#)

[Viewing compilation errors and warnings](#)

■ Related reference

[New Java Scrapbook Page wizard](#)

[Java scrapbook page](#)

[Expressions view](#)

© Copyright IBM Corporation and others 2000, 2004.

Notices

The material in this guide is Copyright (c) IBM Corporation and others 2000, 2005.

[Terms and conditions regarding the use of this guide.](#)

About This Content

February 24, 2005

License

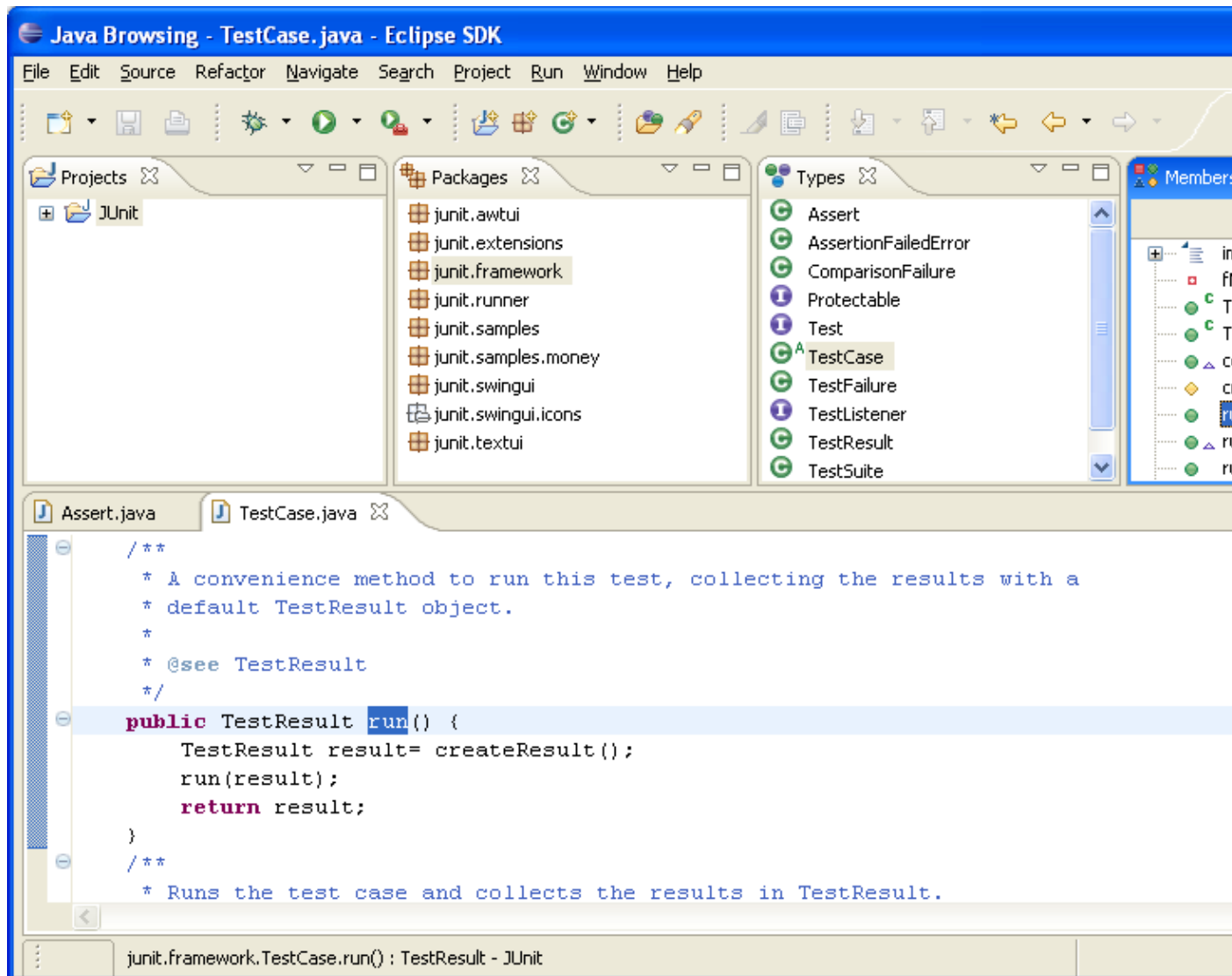
The Eclipse Foundation makes available all content in this plug-in ("Content"). Unless otherwise indicated below, the Content is provided to you under the terms and conditions of the Eclipse Public License Version 1.0 ("EPL"). A copy of the EPL is available at <http://www.eclipse.org/legal/epl-v10.html>. For purposes of the EPL, "Program" will mean the Content.

If you did not receive this Content directly from the Eclipse Foundation, the Content is being redistributed by another party ("Redistributor") and different terms and conditions may apply to your use of any object code in the Content. Check the Redistributor's license that was provided with the Content. If no such license exists, contact the Redistributor. Unless otherwise indicated below, the terms and conditions of the EPL still apply to any source code in the Content.

Using the Java browsing perspective

In this section you will use the Java browsing perspective to browse and manipulate your code. [Browsing Java elements with the Package Explorer](#) gives an overview of using the Package Explorer to browse elements. In contrast to the Package Explorer, which organizes all Java elements in a tree, consisting of projects, packages, compilation units, types, etc., the browsing perspective uses distinct views to present the same information. Selecting an element in one view, will show its content in another view.

To open a browsing perspective activate **Window > Open Perspective > Java Browsing** from within the Java perspective or use the context menu of the *Open a Perspective* toolbar button.



The views of the perspective are connected to each other in the following ways:

- Selecting an element in the **Projects** views shows its packages in the **Packages** view.
- The **Types** view shows the types contained in the package selected in the **Packages** view.
- The **Members** view shows the members of a selected type. Functionally, the **Members** view is comparable to the **Outline** view used in the normal Java perspective.

Basic tutorial

- Selecting an element in the **Members** view reveals the element in the editor. If there isn't an editor open for the element, double-clicking on the element will open a corresponding editor.

All four views are by default linked to the active editor. This means that the views will adjust their content and their selection according to the file presented in the active editor. The following steps illustrate this behavior:

1. Select *junit.extensions* in the **Packages** view.
2. Open type *TestSetup* in the editor by double-clicking it in the **Types view**.
3. Now give back focus to the editor opened on file *TestCase.java* by clicking on the editor tab. The **Packages**, **Types** and **Members** view adjust their content and selections to reflect the active editor. The **Packages** view's selection is set to *junit.framework* and the **Types** view shows the content of the *junit.framework* packages. In addition, the type *TestCase* is selected.

Functionally, the Java browsing perspective is fully comparable to the Java perspective. The context menus for projects, packages, types, etc. and the global menu and tool bar are the same. Therefore activating these functions is analogous to activating them in the Java perspective.

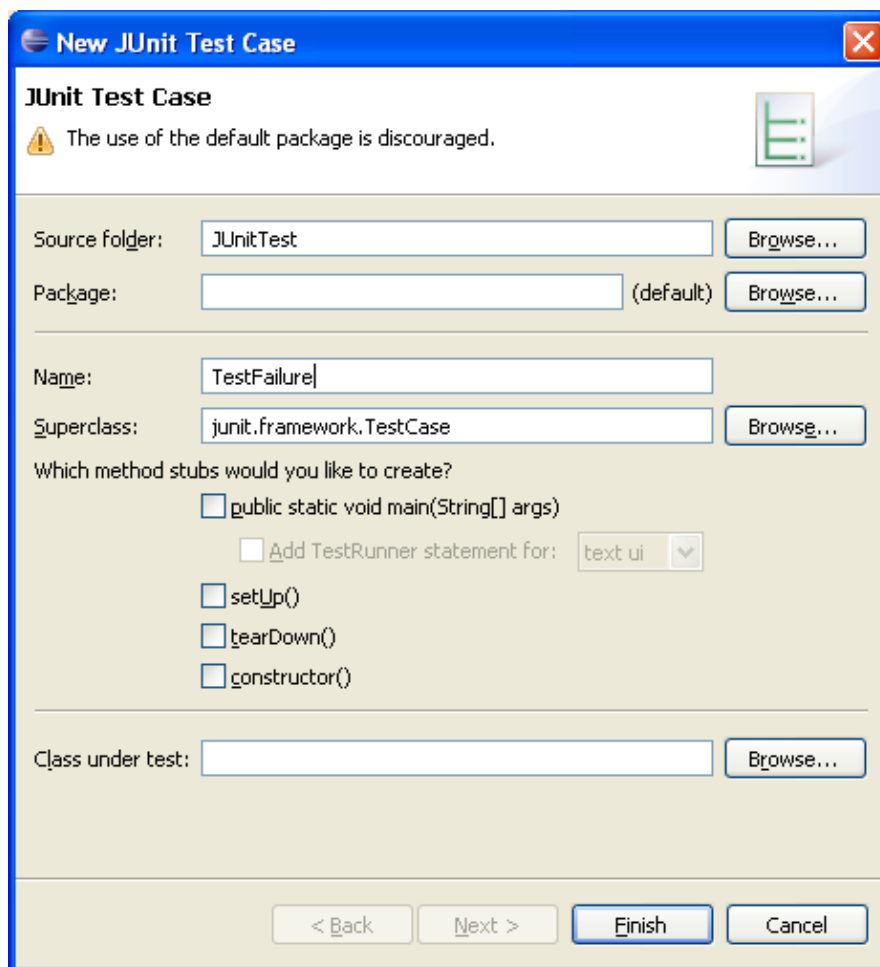
Writing and running JUnit tests

In this section, you will be using the [JUnit](#) testing framework to write and run tests. To get started with JUnit you can refer to the [JUnit Cookbook](#).

Writing Tests

Create a project "JUnitTest". Now you can write your first test. You implement the test in a subclass of *TestCase*. You can do so either using the standard Class wizard or the specialized *Test Case* wizard:

1. Open the New wizard (*File > New > JUnit Test Case*).
2. A dialog will open asking to add the junit library to the class path. Select *Yes*.
3. Enter "TestFailure" as the name of your test class:



4. Click *Finish* to create the test class.

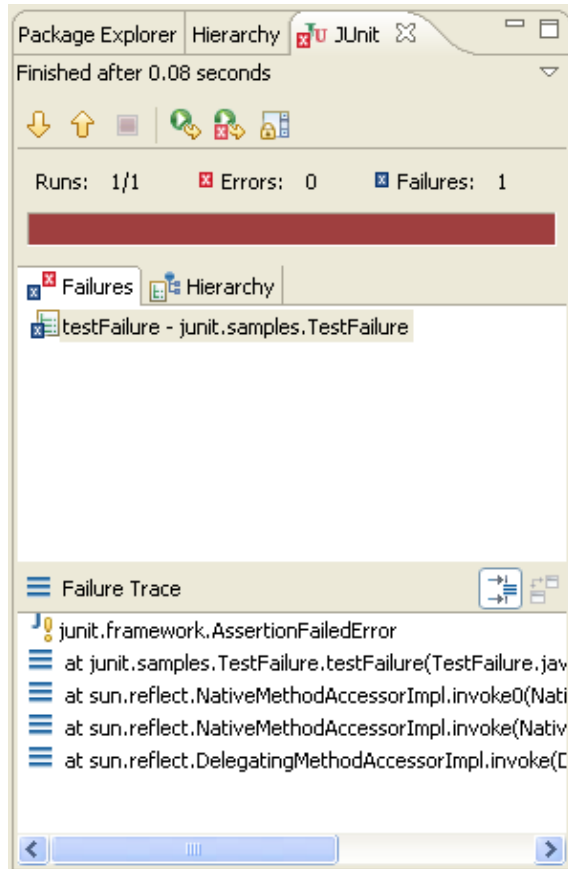
Add a test method that fails to the class *TestFailure*. A quick way to enter a test method is with the *test* template. To do so, place the cursor inside the class declaration. Type "test" followed by Ctrl+Space to activate code assist and select the "test" template. Change the name of the created method to *testFailure* and invoke the *fail()* method.

```
public void testFailure() throws Exception {
    fail();
}
```

Now you are ready to run your first test.

Running Tests

To run `TestFailure`, and activate the **Run** drop-down menu in the toolbar and select **Run as > JUnit Test**. You can inspect the test results in the *JUnit* view. This view shows you the test run progress and status:



The view is shown in the current perspective whenever you start a test run. A convenient arrangement for the JUnit view is to dock it as a fast view. The JUnit view has two tabs: one shows you a list of failures and the other shows you the full test suite as a tree. You can navigate from a failure to the corresponding source by double clicking the corresponding line in the failure trace.

Dock the JUnit view as a fast view, remove the `fail()` statement in the method `testFailure()` so that the test passes and rerun the test again. You can rerun a test either by clicking the **Rerun** button in the view's tool bar or you can re-run the program that was last launched by activating the **Run** drop down. This time the test should succeed. Because the test was successful, the JUnit view doesn't pop up, but the success indicator shows on the JUnit view icon and the status line shows the test result. As a reminder to rerun your tests the view icon is decorated by a "*" whenever you change the workspace contents after the run.



– A successful test run

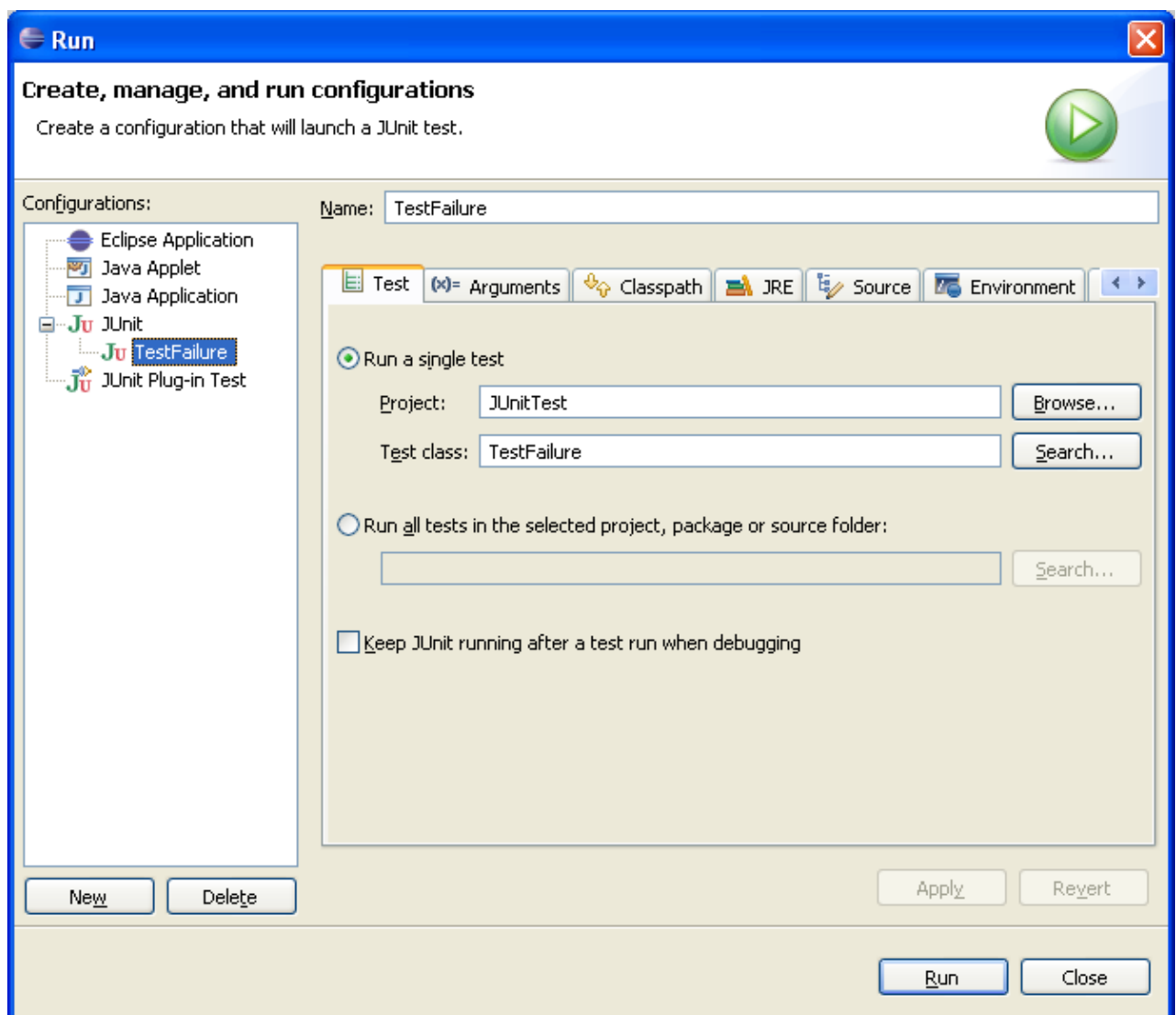
– A successful test run, but the workspace contents have changed since the last test run.

In addition to running a test case as described above you can also:

- Run all tests inside a project, source folder, or package –
Select a project, package or source folder and run all the included tests with **Run as > JUnit Test**. This command finds all tests inside a project, source folder or package and executes them.
- Run a single test method –
Select a test method in the Outline or Package Explorer and with **Run as > JUnit Test** the selected test method will be run.
- Rerun a single test –
Select a test in the JUnit view and execute **Run** from the context menu.

Customizing a Test Configuration

When you want to pass parameters or customize the settings for a test run you open the Launch Configuration Dialog. Select **Run...** in the **Run** drop-down menu in the toolbar:



In this dialog you can specify the test to be run, its arguments, its run-time class path, and the Java run-time environment.

Debugging a Test Failure

In the case of a test failure you can follow these steps to debug it:

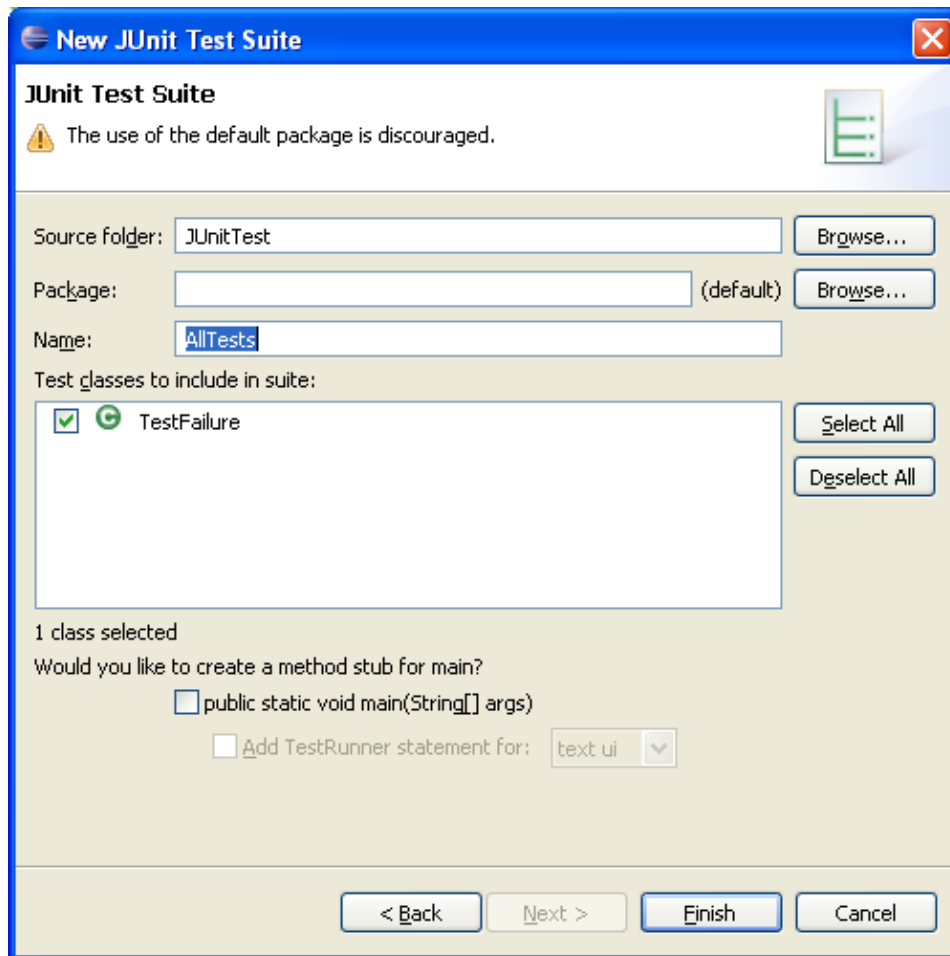
1. Double click the failure entry from the Failures tab in the JUnit view to open the corresponding file in the editor.
2. Set a breakpoint at the beginning of the test method.
3. Select the test case and execute **Debug As>JUnit Test** from the **Debug** drop down.

A JUnit launch configuration has a "keep alive" option. If your Java virtual machine supports "hot code replacement" you can fix the code and rerun the test without restarting the full test run. To enable this option select the *Keep JUnit running after a test run when debugging* checkbox in the JUnit launch configuration.

Creating a Test Suite

The JUnit **TestSuite** wizard helps you with the creation of a test suite. You can select the set of classes that should belong to a suite.

1. Open the New wizard
2. Select *Java > JUnit > JUnit Test Suite* and click *Next*.
3. Enter a name for your test suite class (the convention is to use "AllTests" which appears by default).



4. Select the classes that should be included in the suite. We currently have a single test class only, but you can add to the suite later.

You can add or remove test classes from the test suite in two ways:

- Manually by editing the test suite file
- By re-running the wizard and selecting the new set of test classes.

Note: the wizard puts 2 markers, `//$JUnit-BEGIN$` and `//$JUnit-END$`, into the created Test suite class, which allows the wizard to update existing test suite classes. Editing code between the markers is not recommended.

Project configuration tutorial

In this section, you will create and configure a new Java project to use source folders and to match some existing layout on the file system. Some typical layouts have been identified. Choose the sub–section that matches your layout.

■ Related concepts

[Java projects](#)

[Java views](#)

■ Related tasks

[Working with build paths](#)

[Creating a new Java project](#)

[Creating a Java project with source folders](#)

[Creating a new source folder](#)

[Using the Package Explorer](#)

■ Related reference

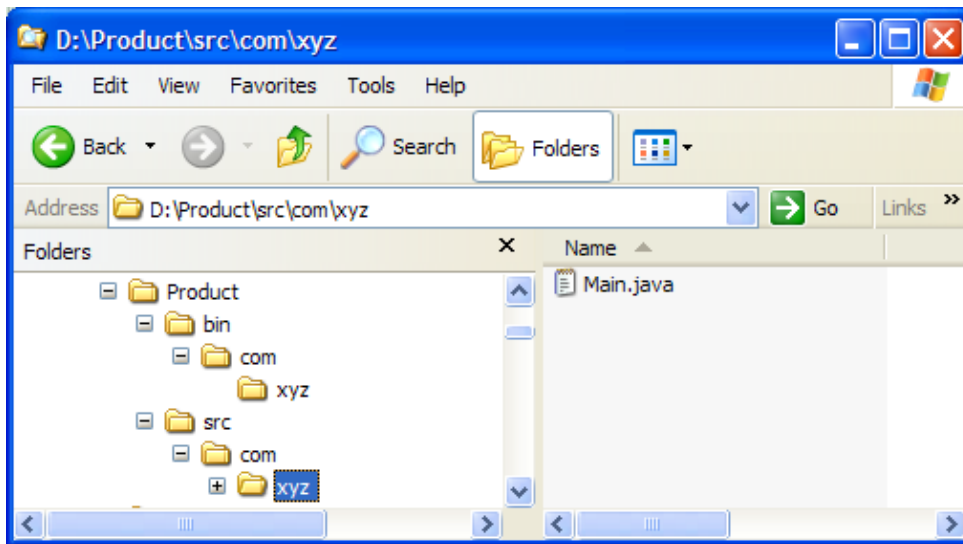
[New Java Project Wizard](#)

[Package Explorer View](#)

Detecting existing layout

Layout on file system

- The source files for a product are laid out in one directory "src".
- The class files are in another directory "bin".



Steps for defining a corresponding project

1. Open a Java perspective, select the menu item **File > New > Project....** to open the *New Project* wizard.
2. Select **Java project** in the list of wizards and click *Next*.
3. On the next page, type "Product" in the *Project name* field.
4. In *Project layout* group, change selection to **Create separate source and output folders**.

In *Contents* group, change selection to **Create project from existing source**.

Click **Browse...** and choose the *Product* directory on drive *D* :.

New Java Project

Create a Java project
Create a Java project in the workspace or in an external location.

Project name:

Contents

☐ Create new project in workspace
☒ Create project from existing source

Directory:

JDK Compliance

☒ Use default compiler compliance (Currently 1.4) [Configure default...](#)
☐ Use a project specific compliance:

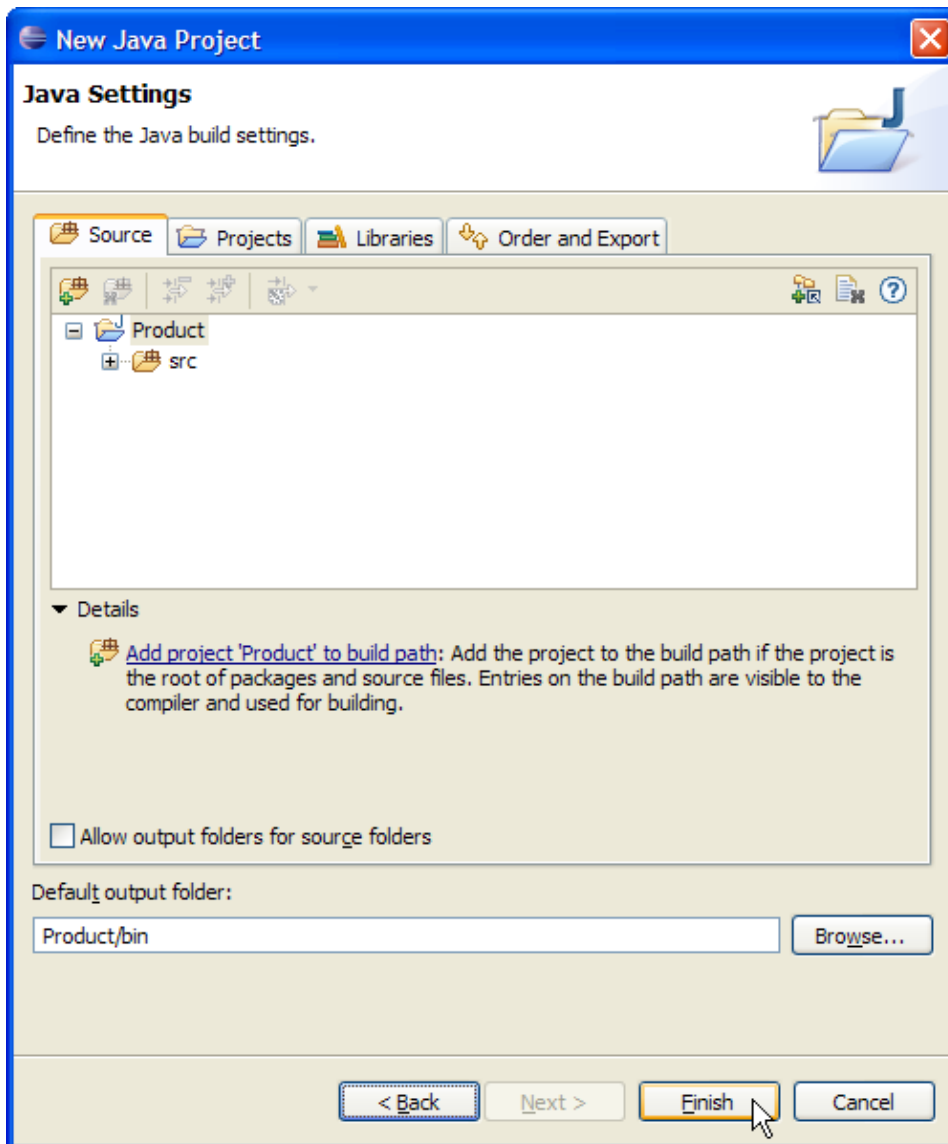
Project layout

☐ Use project folder as root for sources and class files
☒ Create separate source and output folders [Configure default...](#)

The specified external location already exists. If a project is created in this location, the wizard will automatically try to detect existing sources and class files and configure the classpath appropriately.

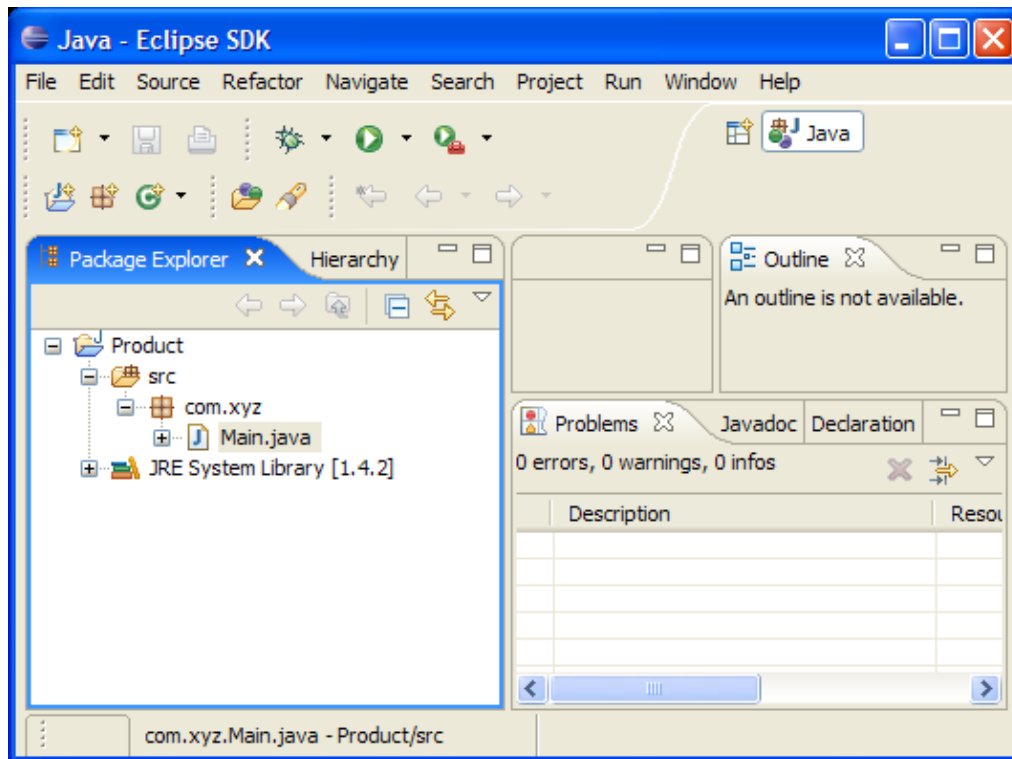
5. Click *Next*.

Ensure that the source and output folders are detected.



Warning: If the preference *Window > Preferences > Java > Compiler > Building > Output Folder > Scrub output folders when cleaning projects* is checked, clicking *Finish* will scrub the "bin" directory in the file system before generating the class files.

6. Click **Finish**.
7. You now have a Java project with a "src" folder which contains the sources of the "Product" directory.



Note: This solution creates a ".project" file and a ".classpath" file in the "Product" directory. If you do not wish to have these files in the "Product" directory, you should use linked folders as shown in the [Sibling products in a common source tree](#) section.

■ Related concepts

[Java projects](#)

[Java views](#)

■ Related tasks

[Working with build paths](#)

[Creating a new Java project](#)

[Creating a Java project with source folders](#)

[Creating a new source folder](#)

[Using the Package Explorer](#)

■ Related reference

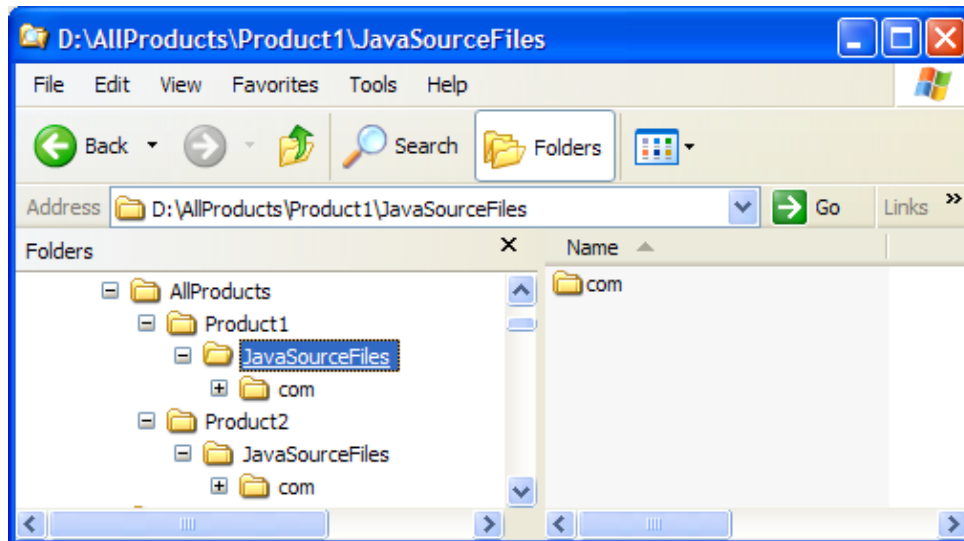
[New Java Project Wizard](#)

[Package Explorer View](#)

Sibling products in a common source tree

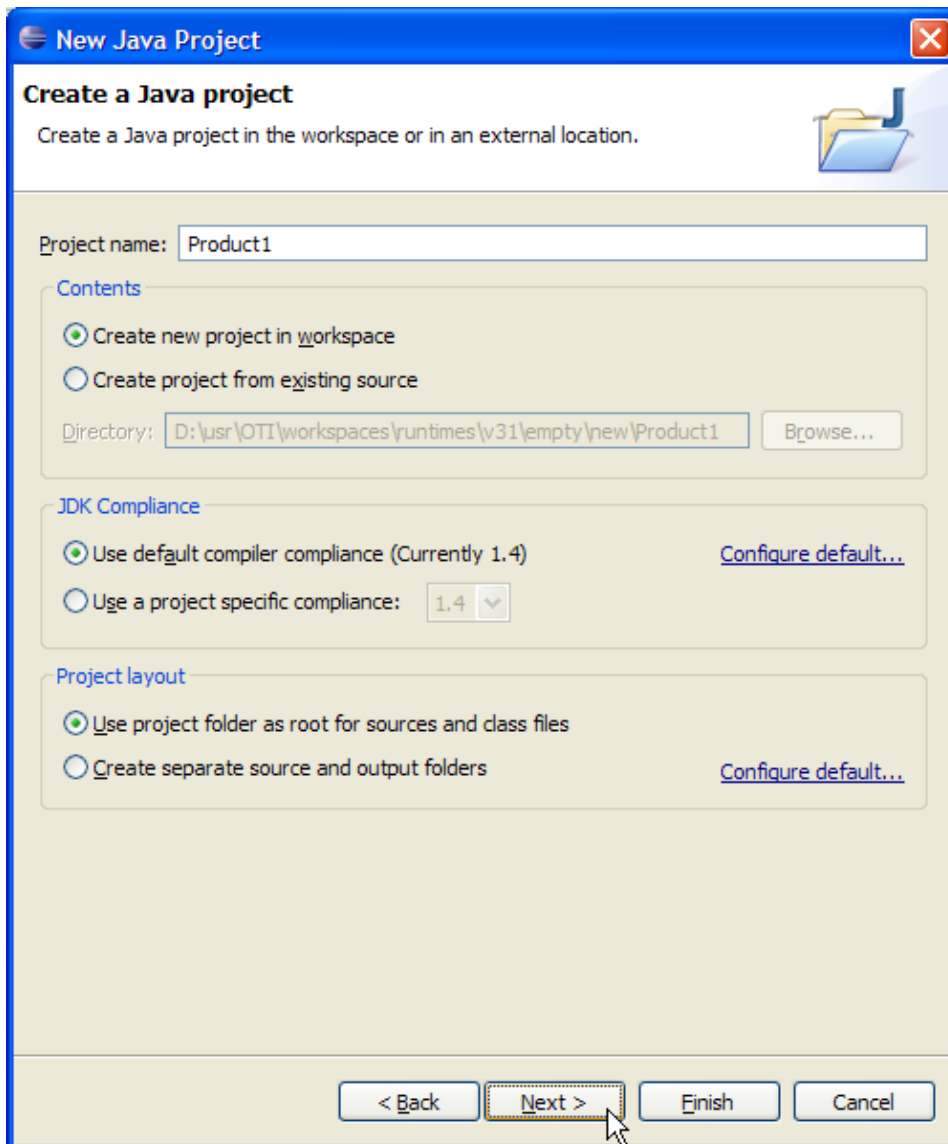
Layout on file system

- The source files for products are laid out in one big directory that is version and configuration managed outside Eclipse.
- The source directory contains two siblings directories *Product1* and *Product2*.




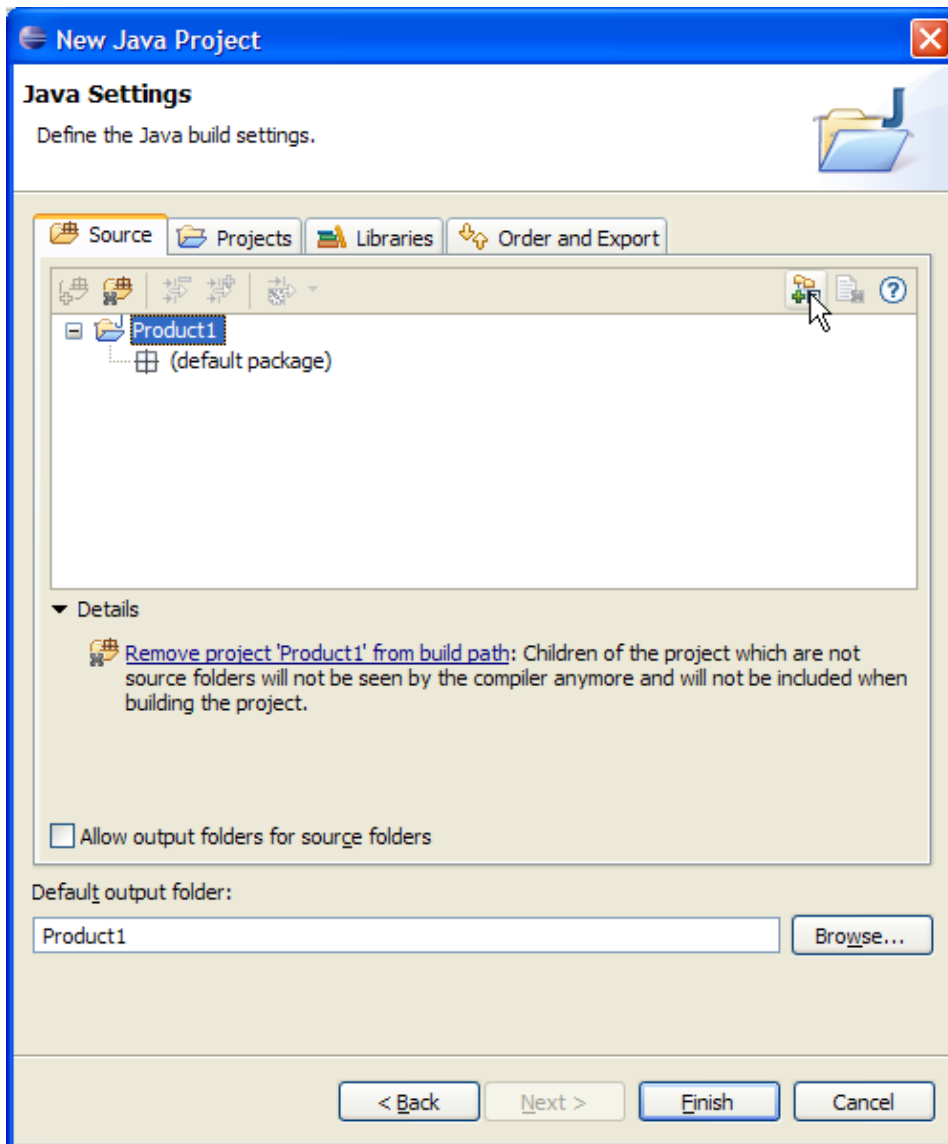
Steps for defining corresponding projects

1. Open a Java perspective, select the menu item **File > New > Project...** to open the *New Project* wizard.
2. Select **Java project** in the list of wizards and click *Next*.
3. On the next page, type "Product1" in the *Project name* field. Click *Next*.

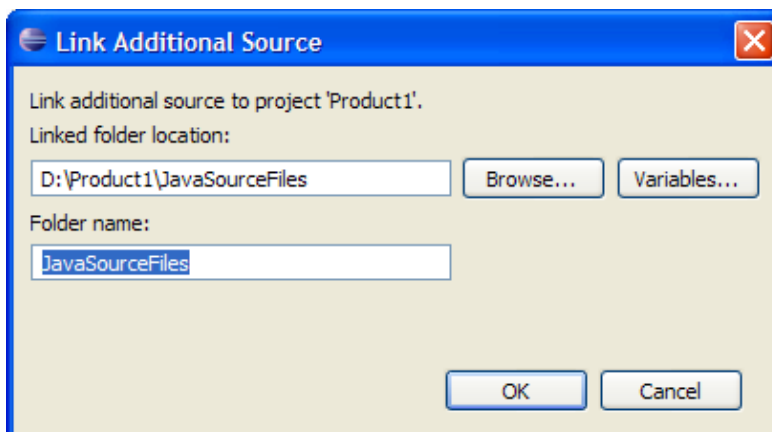


4. On the next page, Select "Product1" source folder.

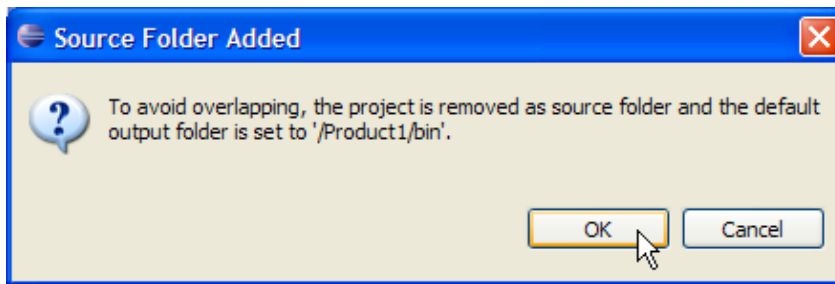
Click **Link Additional Source to Project** button  in view bar.



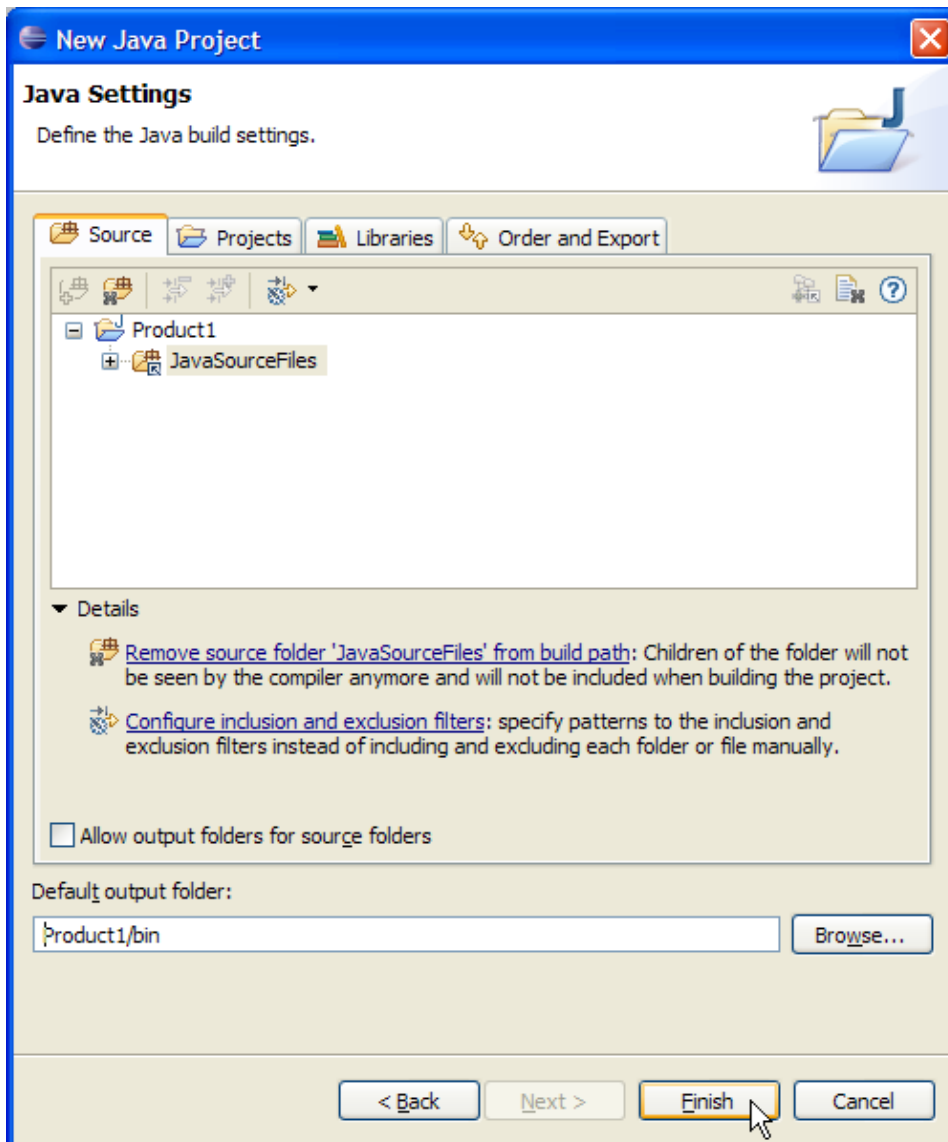
5. In **Link Additional Source** click **Browse....** and choose the `D:\Product1\JavaSourceFiles` directory.



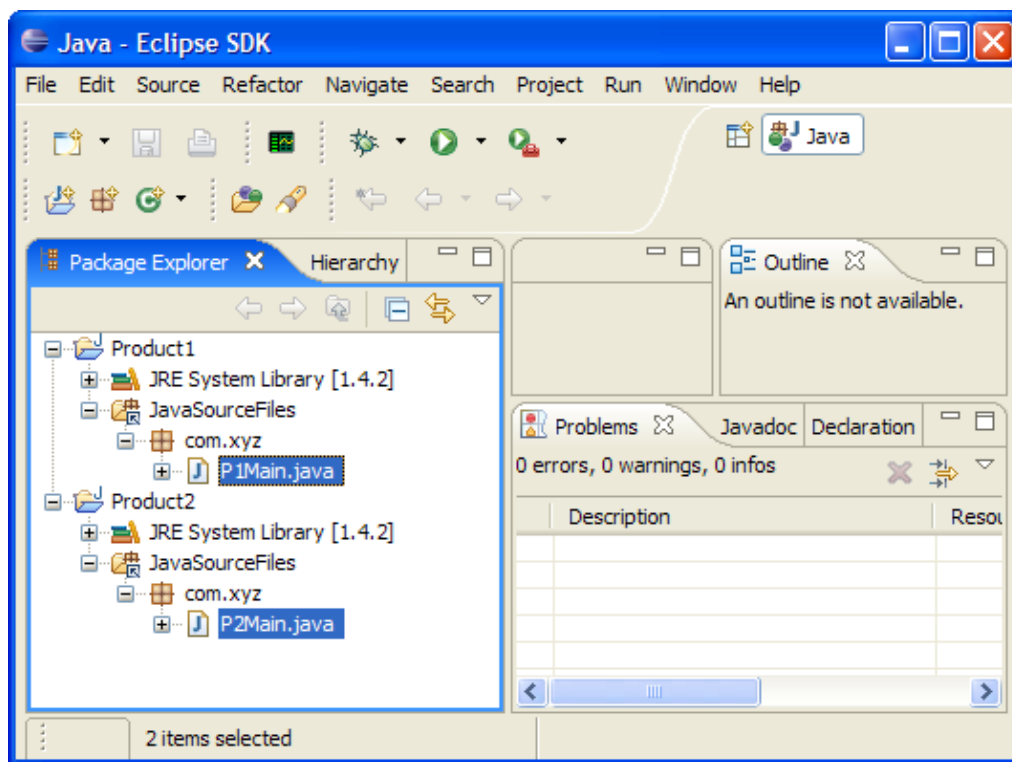
6. Click **OK** to close the dialog.
 7. Click **OK** in confirmation dialog to have `"/Product1/bin"` as default output folder.



8. Your project source setup now looks as follows:



9. Click **Finish**.
10. Repeat these steps for "Product2".
11. You now have two Java projects which respectively contain the sources of "Product1" and "Product2".



■ Related concepts

[Java projects](#)

[Java views](#)

■ Related tasks

[Working with build paths](#)

[Creating a new Java project](#)

[Creating a Java project with source folders](#)

[Creating a new source folder](#)

[Using the Package Explorer](#)

■ Related reference

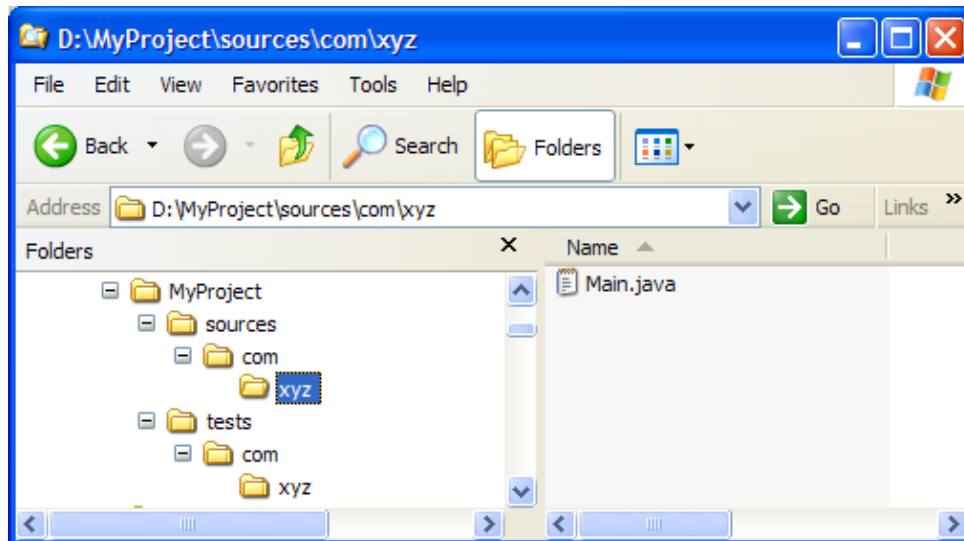
[New Java Project Wizard](#)

[Package Explorer View](#)

Organizing sources

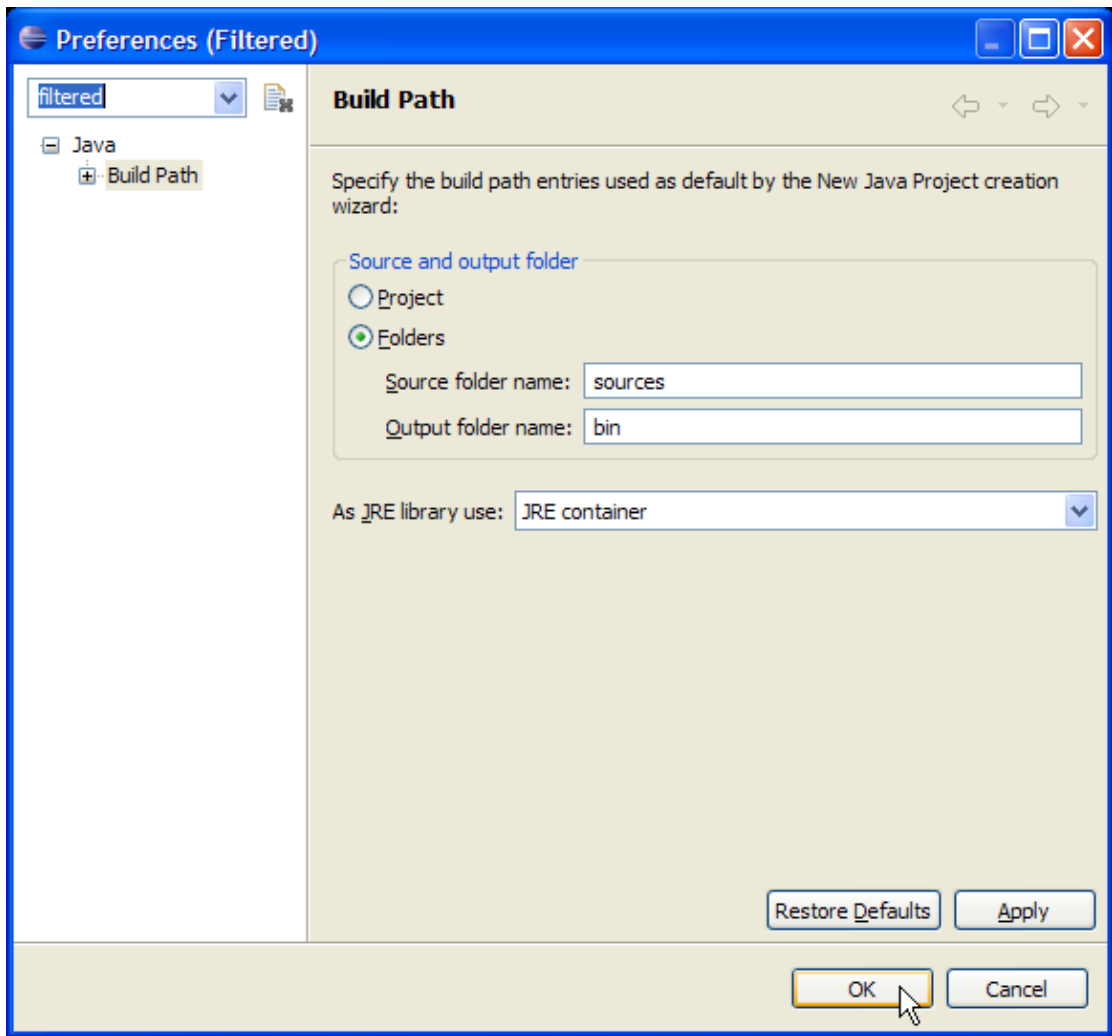
Layout on file system

- In this section, you will create a new Java project and organize your sources in separate folders. This will prepare you for handling more complex layouts.
- Let's assume you want to put your sources in one folder and your tests in another folder:

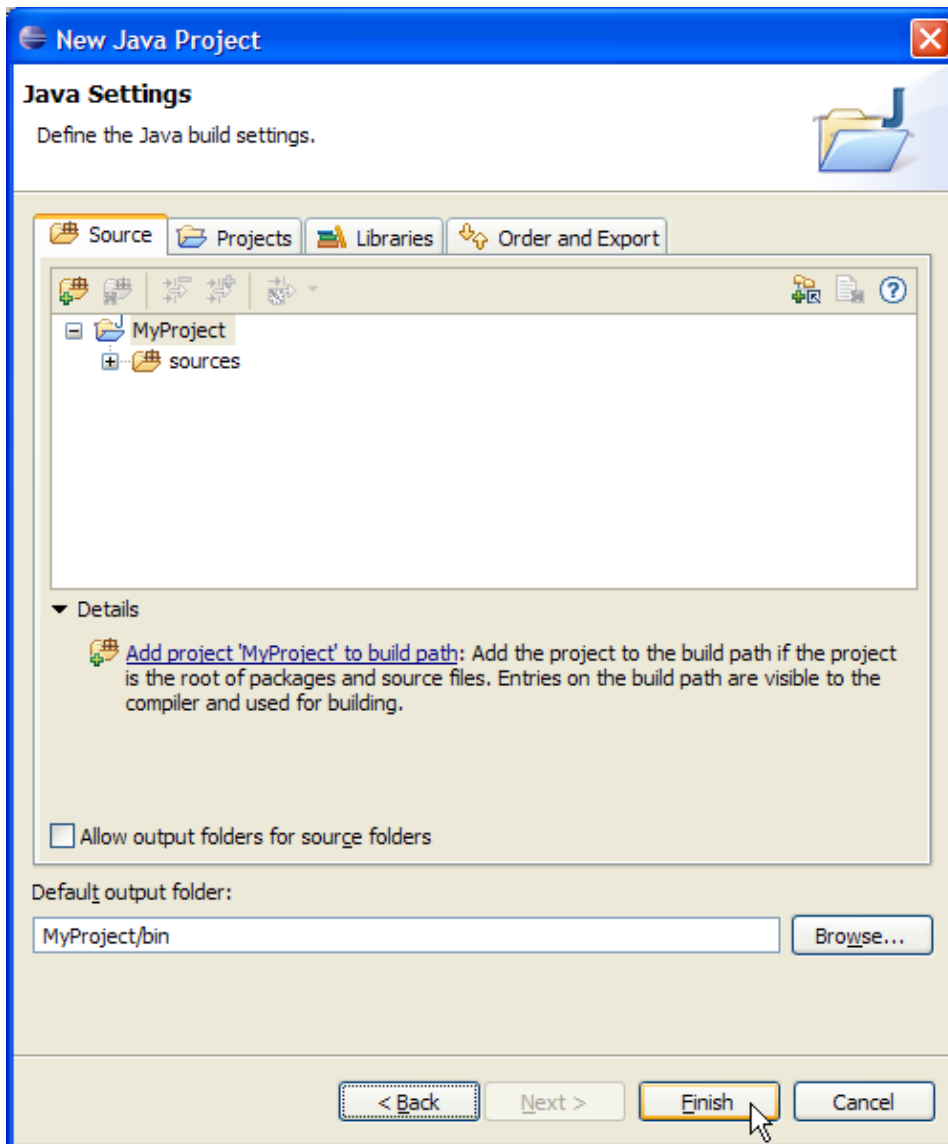


Steps for defining a corresponding project

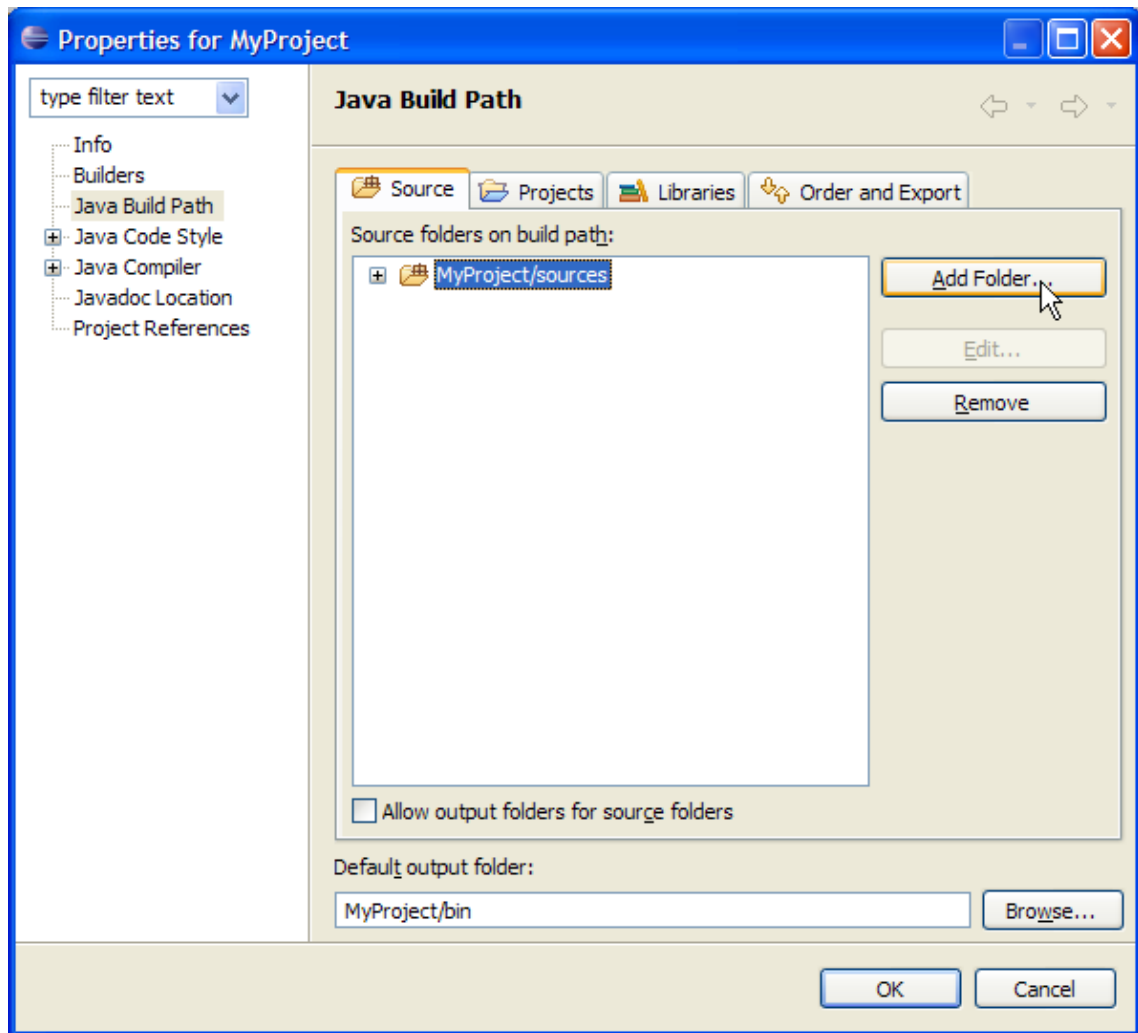
1. Open a Java perspective, select the menu item **File > New > Project...** to open the *New Project* wizard.
2. Select **Java project** in the list of wizards and click *Next*.
3. On the next page, type "MyProject" in the **Project name** field.
4. In **Project layout** group, change selection to **Create separate source and output folders** and edit *Configure default...* to modify **Source folder name** from "src" to "sources".



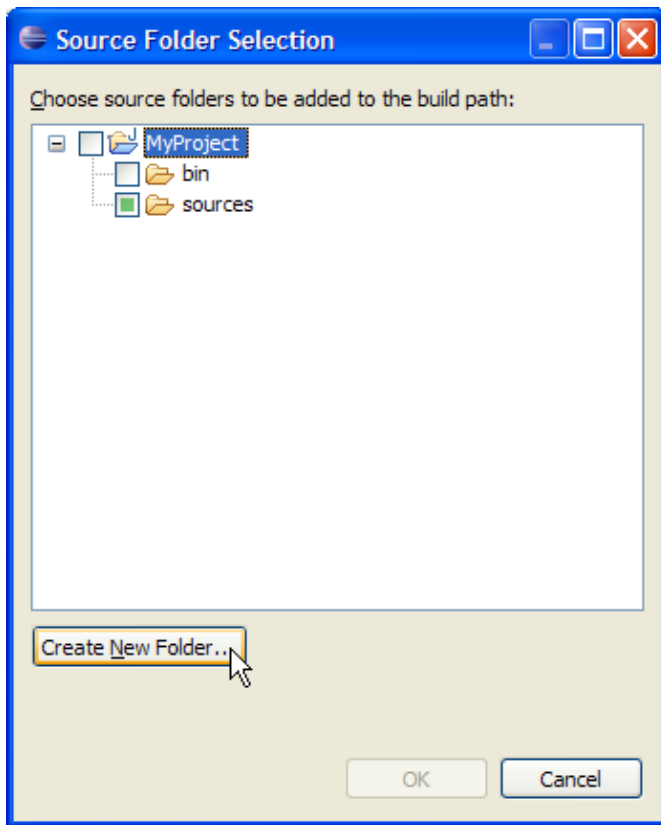
5. Click **OK** to return on New Java Project wizard and then click **Next**.



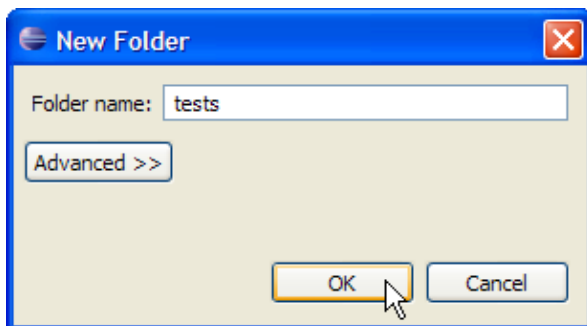
6. Click **Finish**
 7. Edit project "MyProject" properties and select **Java Builder Path** page.
- On **Source** tab, click **Add Folder....**



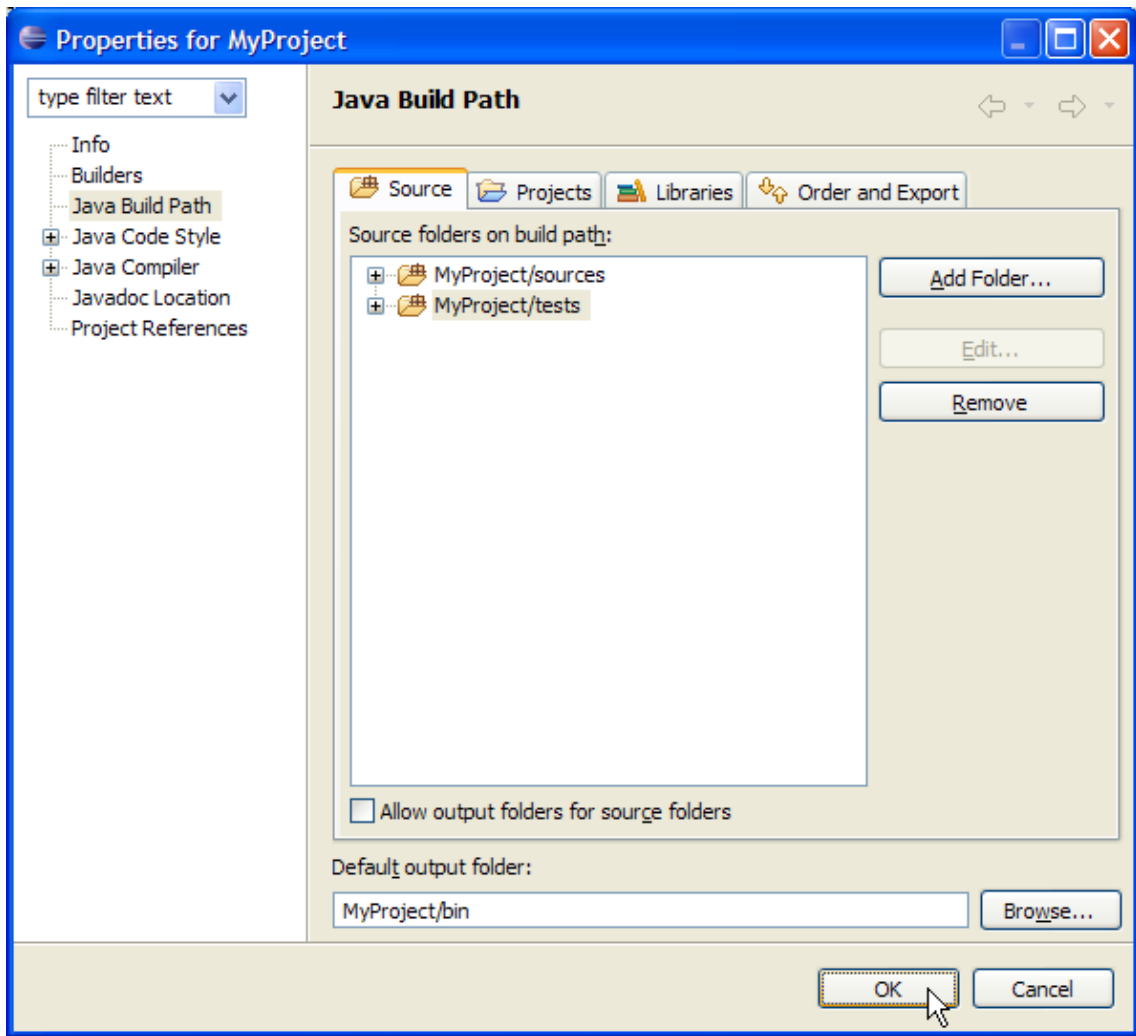
8. On *Source Folder Selection* click *Create New Folder...*



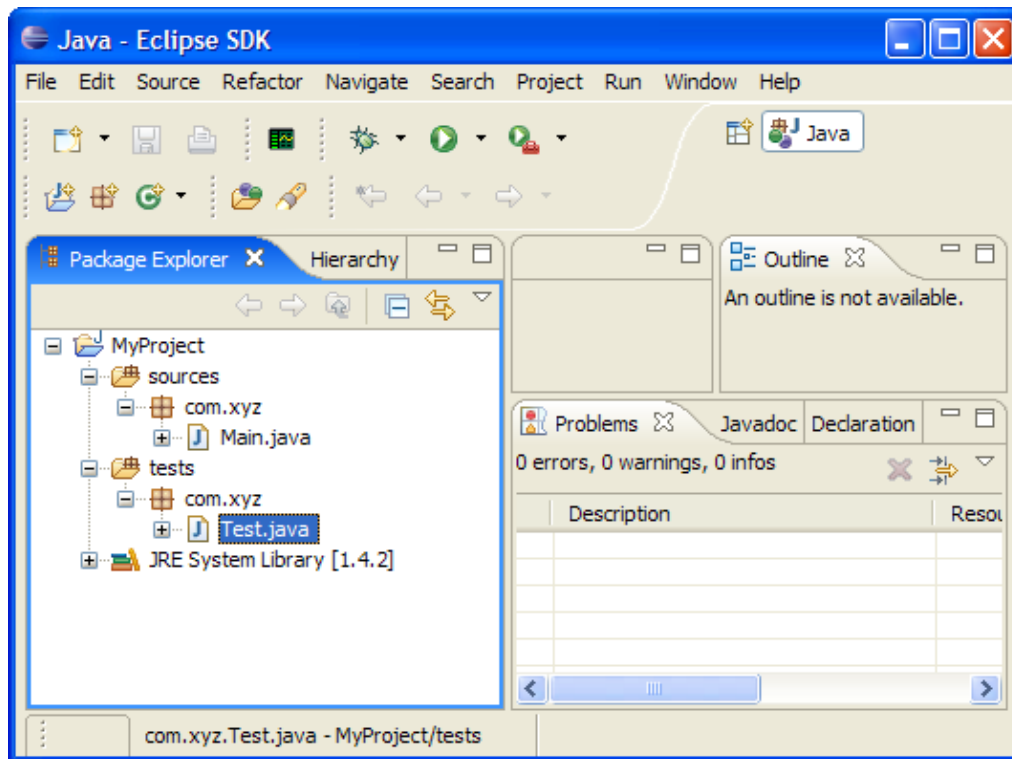
9. In *New Folder* dialog, type "tests" in the *Folder name* field.



10. Click **OK** to close the dialog.
11. Your project setup now looks as follows:



12. Click **OK**
13. You now have a Java project with a *sources* and a *tests* folders. You can start adding classes to these folders or you can copy them using drag and drop.



■ Related concepts

[Java projects](#)

[Java views](#)

■ Related tasks

[Working with build paths](#)

[Creating a new Java project](#)

[Creating a Java project with source folders](#)

[Creating a new source folder](#)

[Using the Package Explorer](#)

■ Related reference

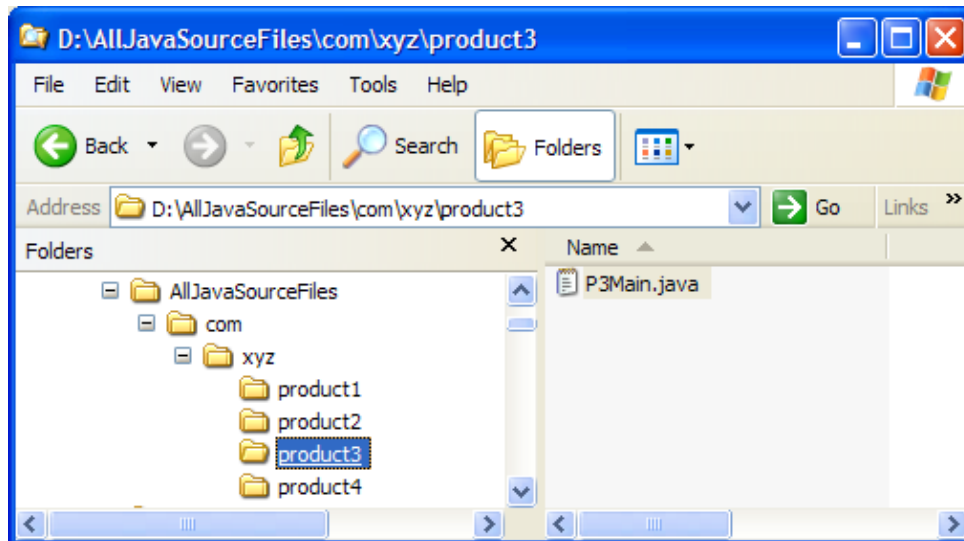
[New Java Project Wizard](#)

[Package Explorer View](#)

Overlapping products in a common source tree

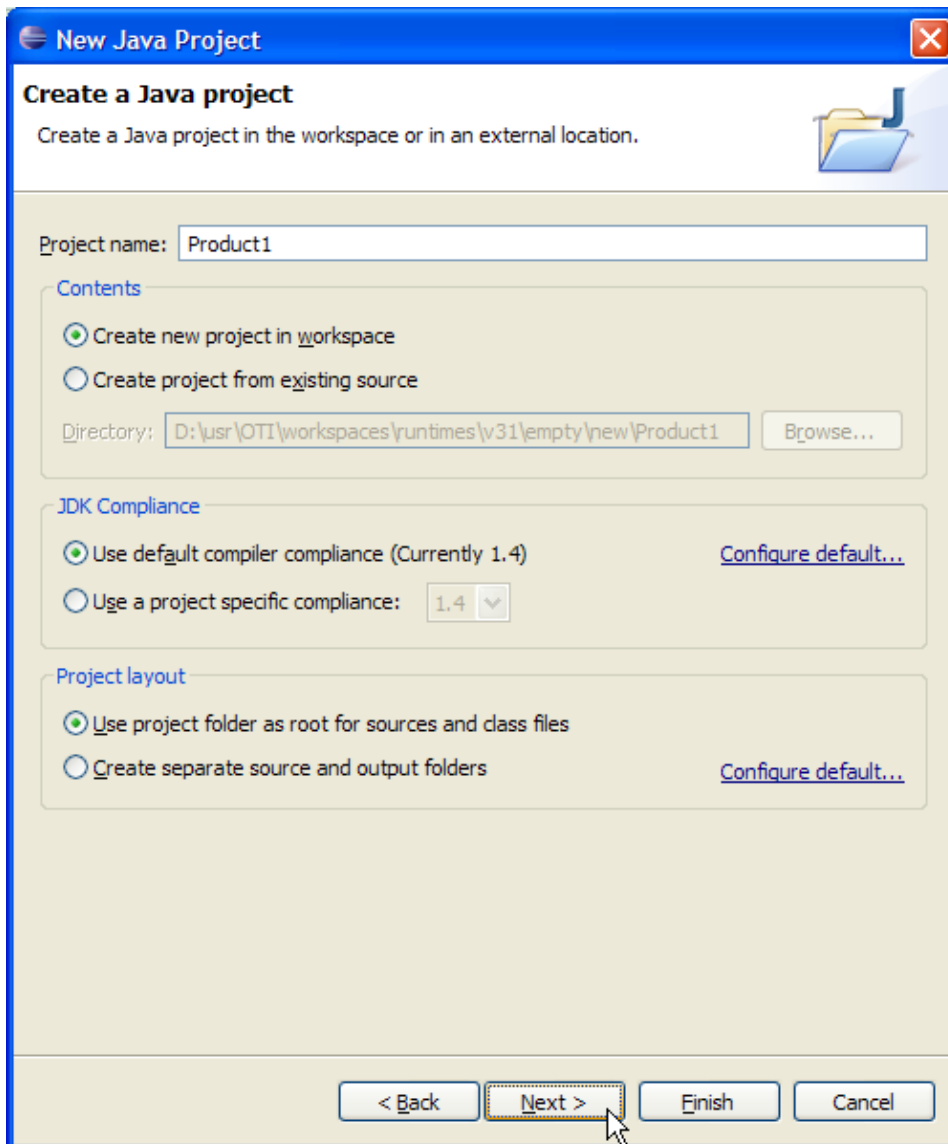
Layout on file system

- The Java source files for products are all held in a single main directory.
- Products are separated into four siblings packages *Product1*, *Product2*, *Product3* and *Product4*.




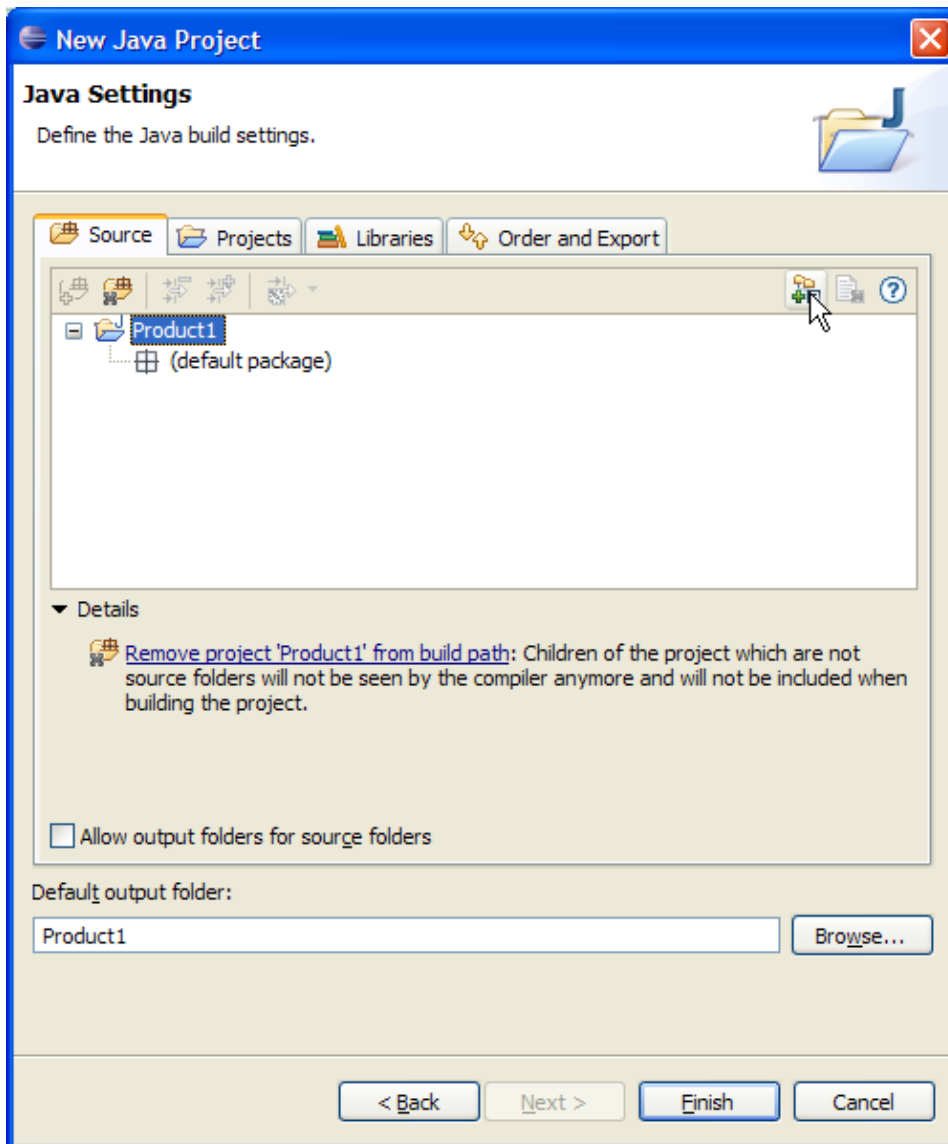
Steps for defining corresponding "Product1" and "Product2" projects

1. Open a Java perspective, select the menu item **File > New > Project....** to open the *New Project* wizard.
2. Select **Java project** in the list of wizards and click *Next*.
3. On the next page, type "Product1" in the *Project name* field. Click *Next*.



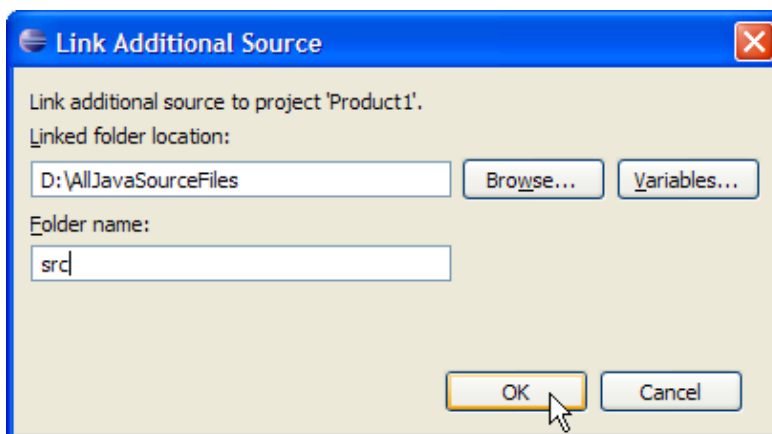
4. On the next page, Select "Product1" source folder.

Click **Link Additional Source to Project** button  in view bar.



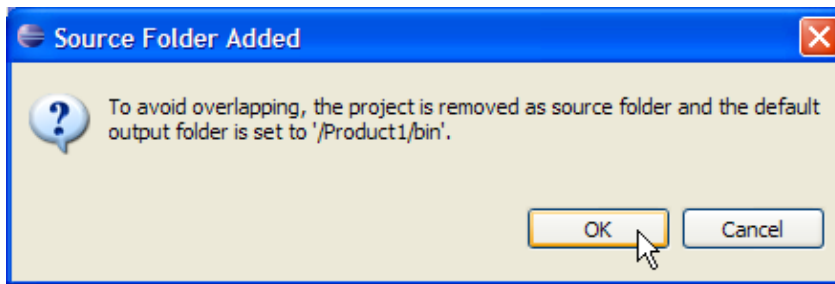
5. In **Link Additional Source** click **Browse....** and choose the `D:\AllJavaSourceFiles` directory.

Type "src" in **Folder name**.

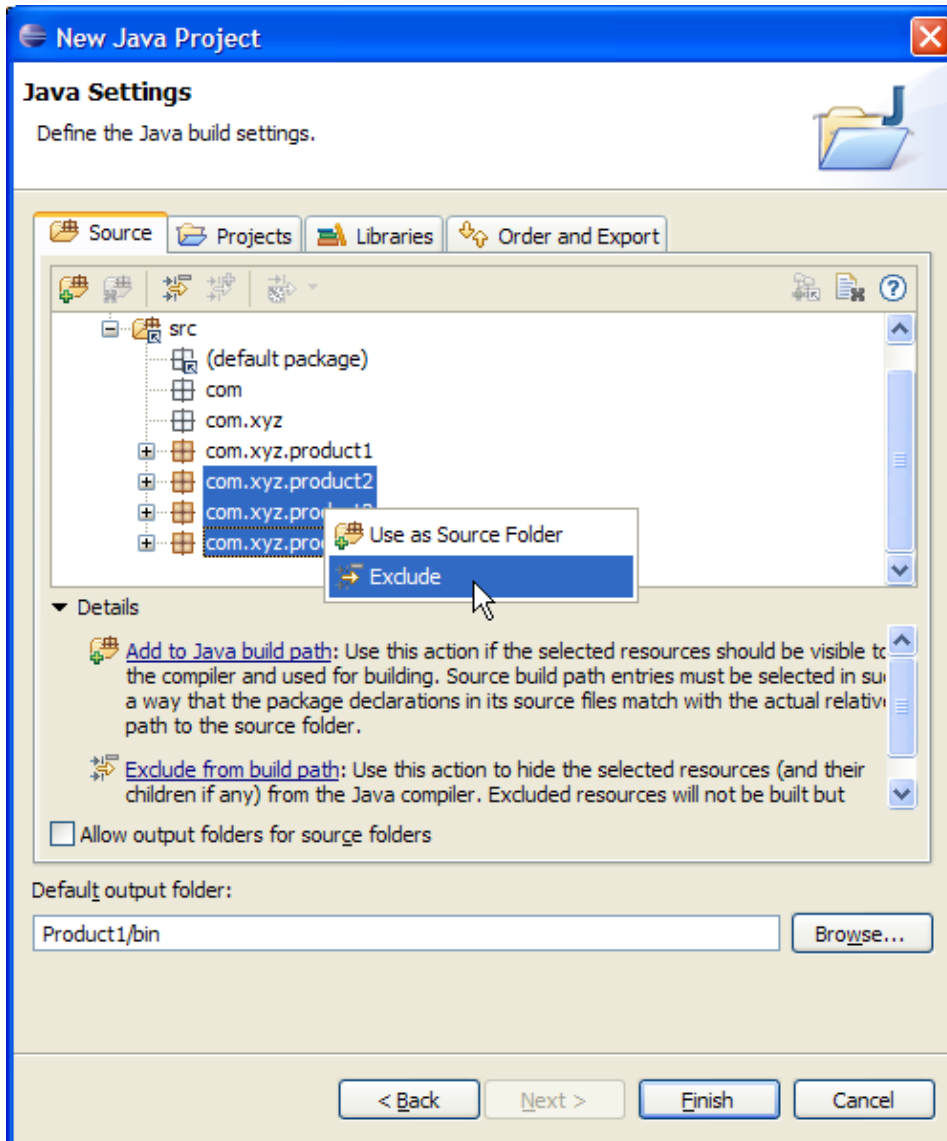


6. Click **OK** to close the dialog.

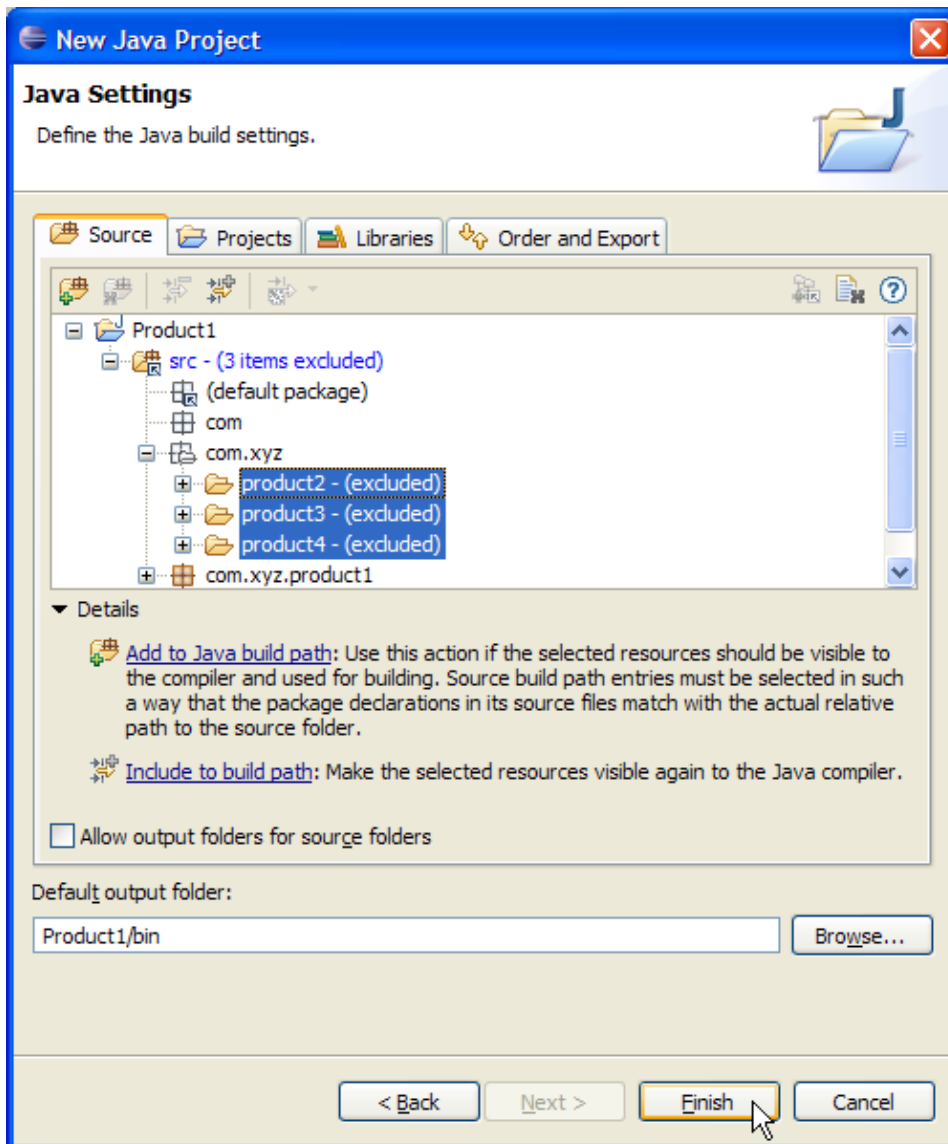
7. Click **OK** in confirmation dialog to have "Product1/bin" as default output folder.



8. Expand the "src" source folder. Select all packages you want to exclude and exclude them using popup-menu.



9. Your project source setup now looks as follows:

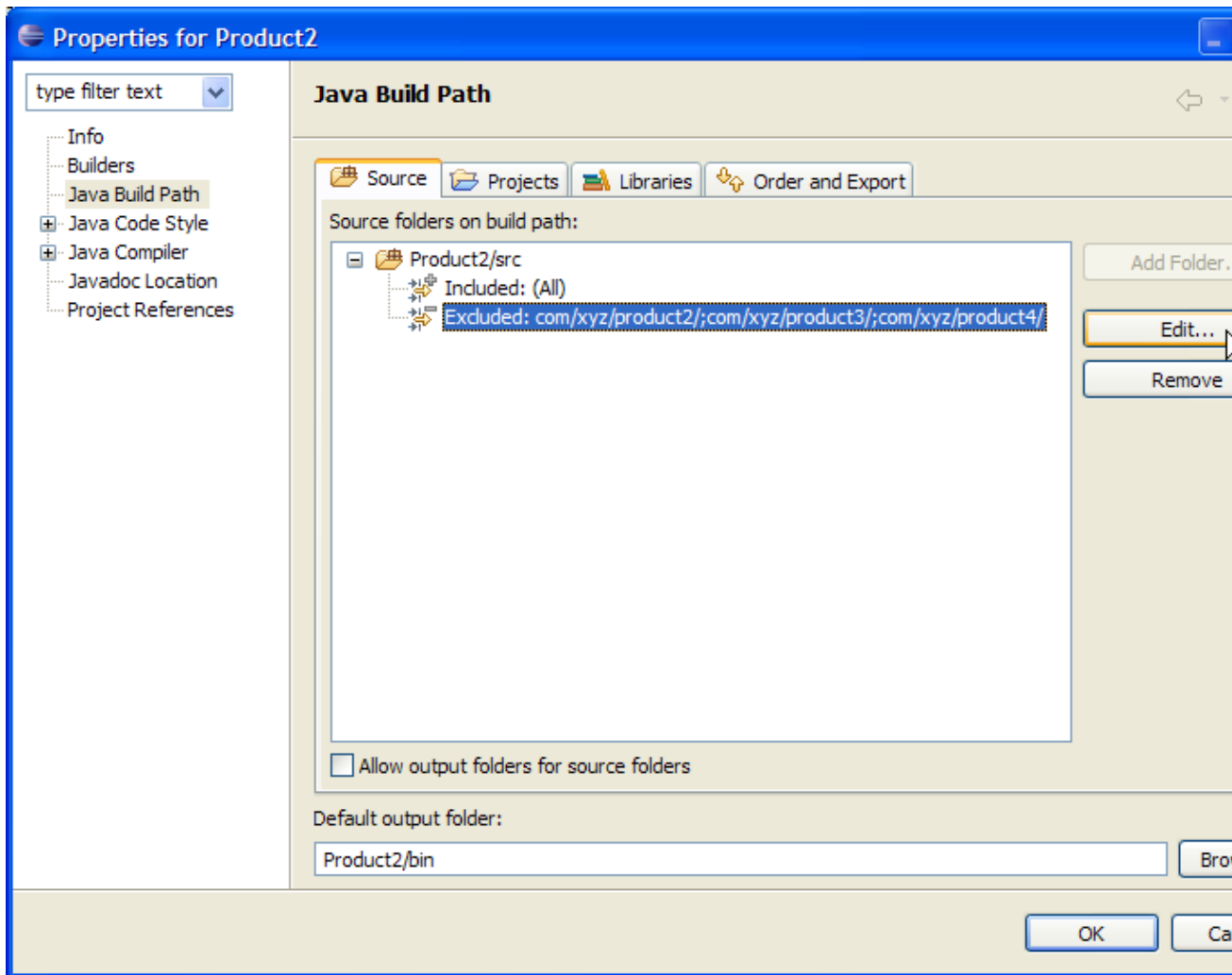


10. Click **Finish**.

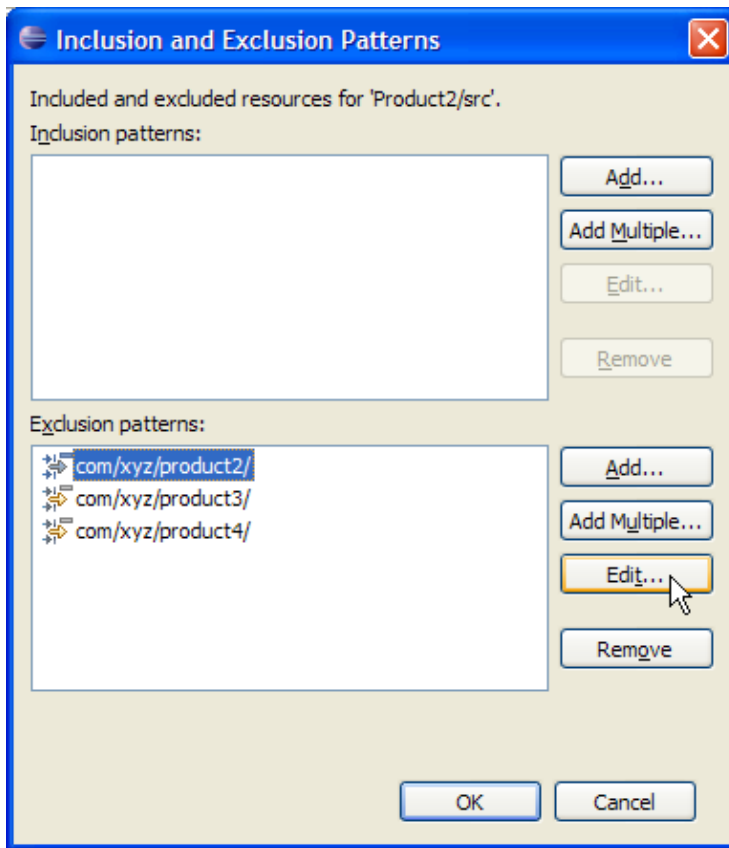
11. Copy "Product1" project and paste it as "Product2".

Edit "Product2" project properties and go on **Java Build Path** page.

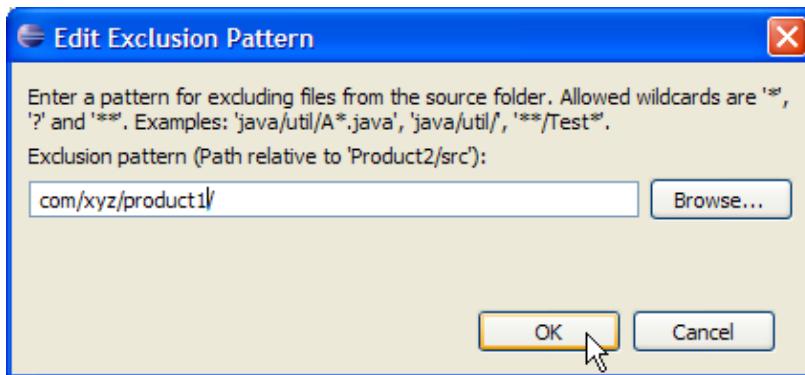
Select **Excluded** and click **Edit...**



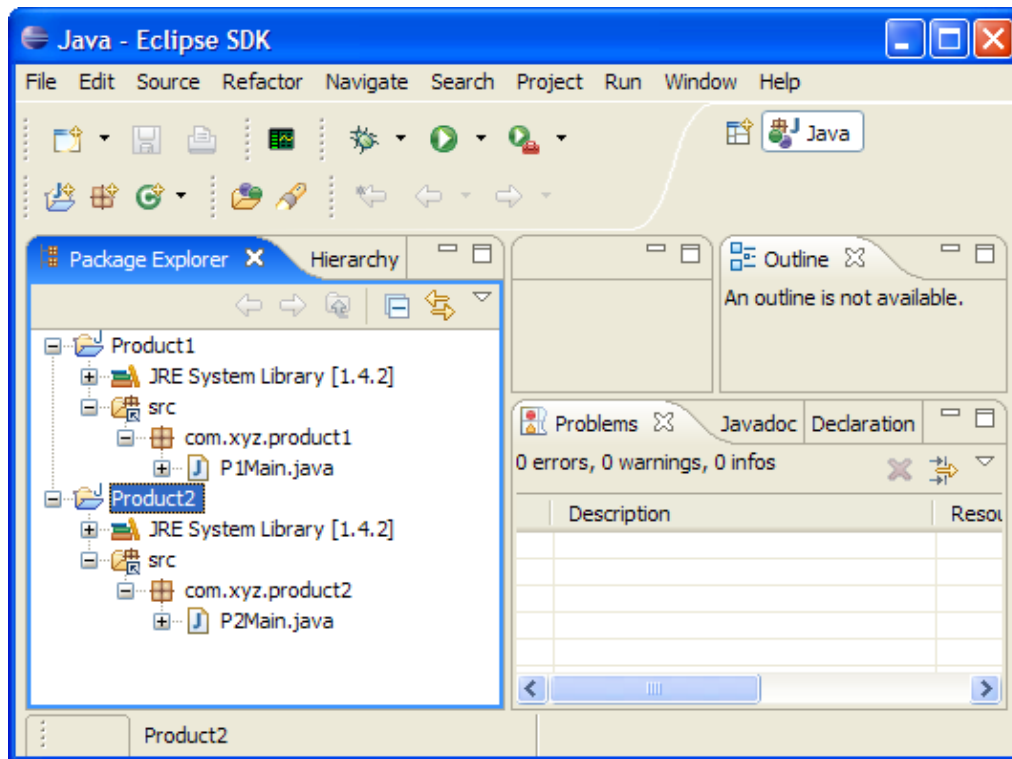
12. In *Inclusion and Exclusion Patterns*, select "com/xyz/product2" and click *Edit...*



13. Change "com/xyz/product2" to "com/xyz/product1" instead.



14. Click **OK** three times to valid and close all dialogs.
15. You now have two Java projects which respectively contain the sources of "product1", "product2".



■ Related concepts

[Java projects](#)

[Java views](#)

■ Related tasks

[Working with build paths](#)

[Creating a new Java project](#)

[Creating a Java project with source folders](#)

[Creating a new source folder](#)

[Using the Package Explorer](#)

■ Related reference

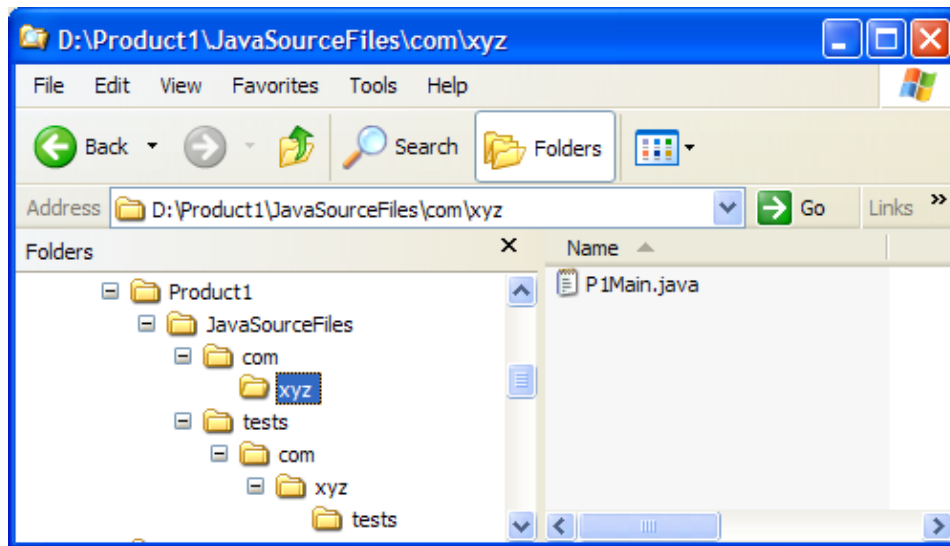
[New Java Project Wizard](#)

[Package Explorer View](#)

Product with nested tests

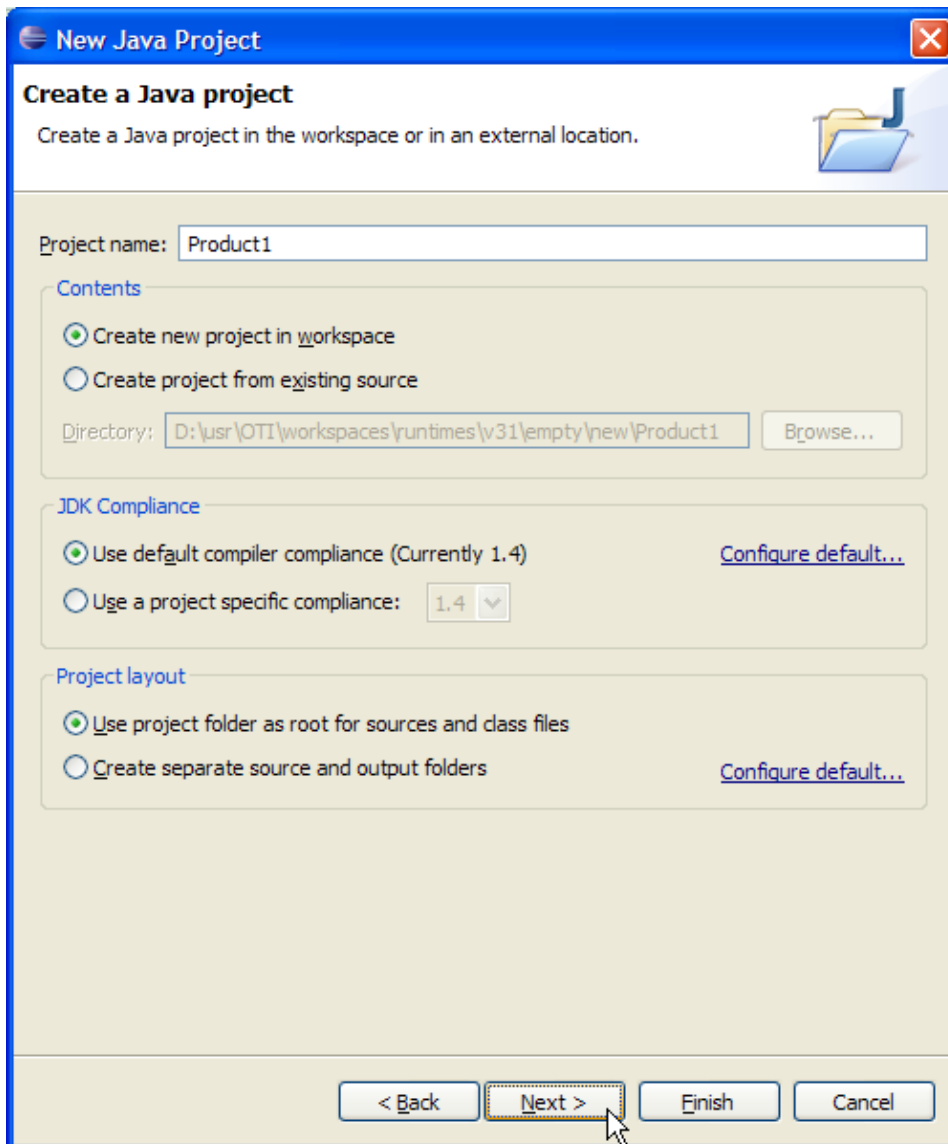
Layout on file system

- The Java source files for a product are laid out in a package directory.
- Source files of tests are laid out in a nested package directory.




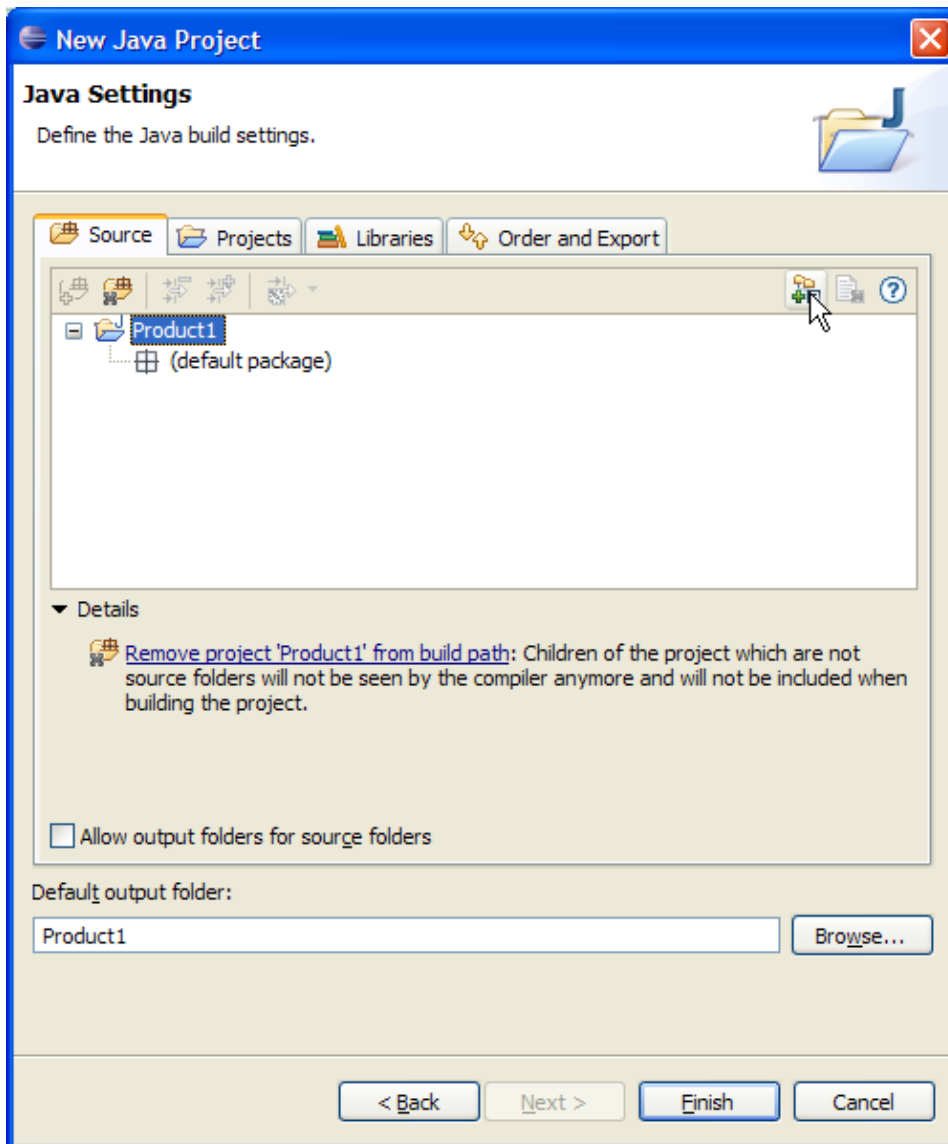
Steps for defining a corresponding project

1. Open a Java perspective, select the menu item **File > New > Project....** to open the *New Project* wizard.
2. Select **Java project** in the list of wizards and click *Next*.
3. On the next page, type "Product1" in the *Project name* field. Click *Next*.



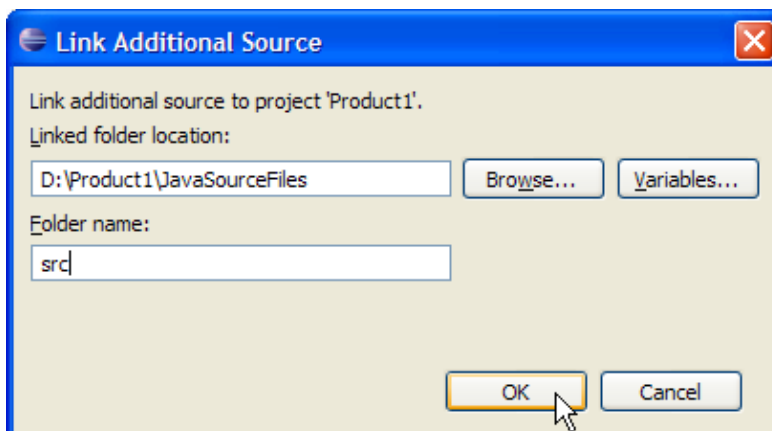
4. On the next page, Select "Product1" source folder.

Click **Link Additional Source to Project** button  in view bar.



5. In **Link Additional Source** click **Browse....** and choose the `D:\Product1\JavaSourceFiles` directory.

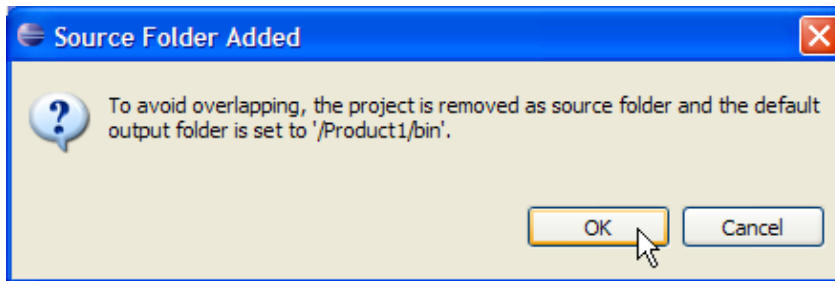
Type "src" in the **Folder name** field.



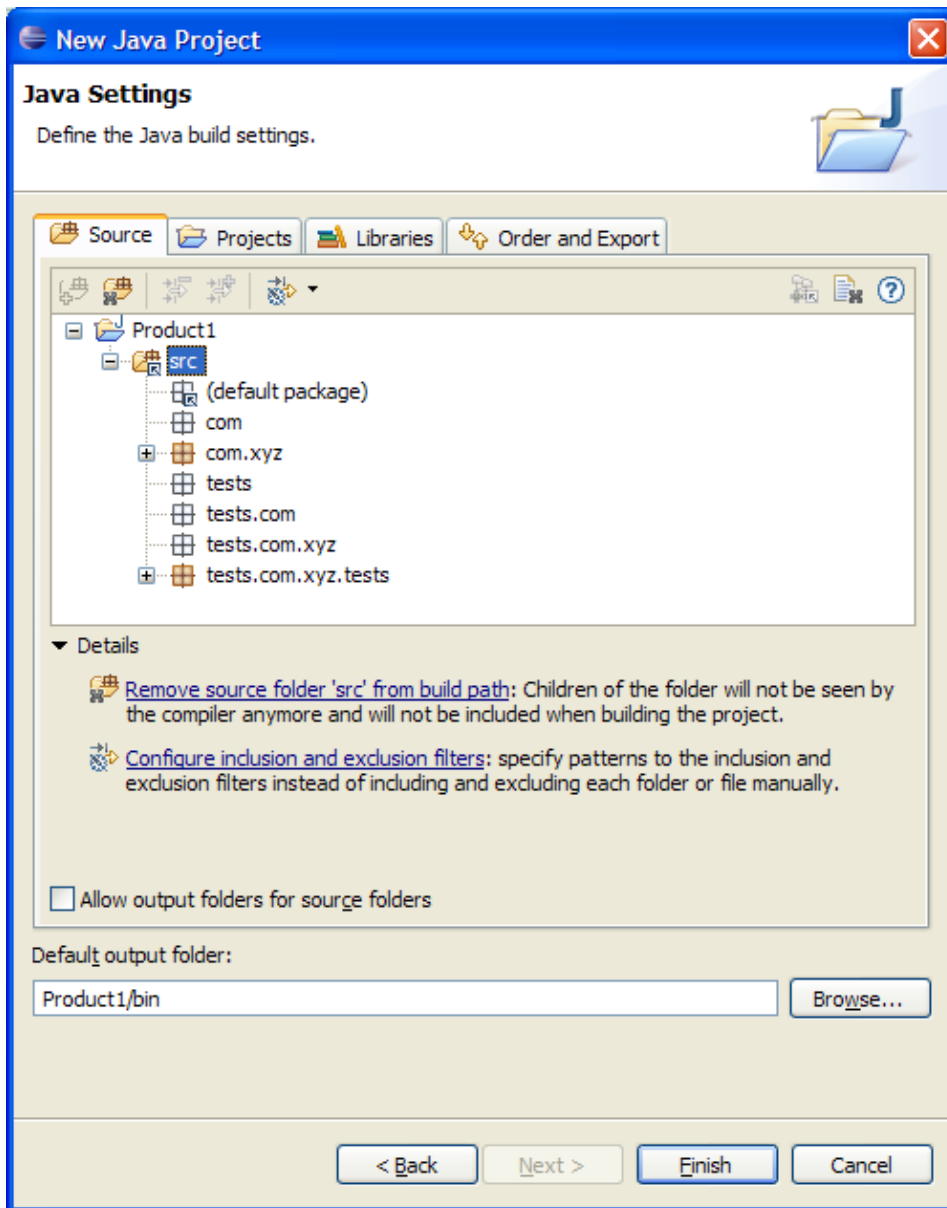
6. Click **OK** to close the dialog.

Basic tutorial

- Click **OK** in confirmation dialog to have "Product1/bin" as default output folder.

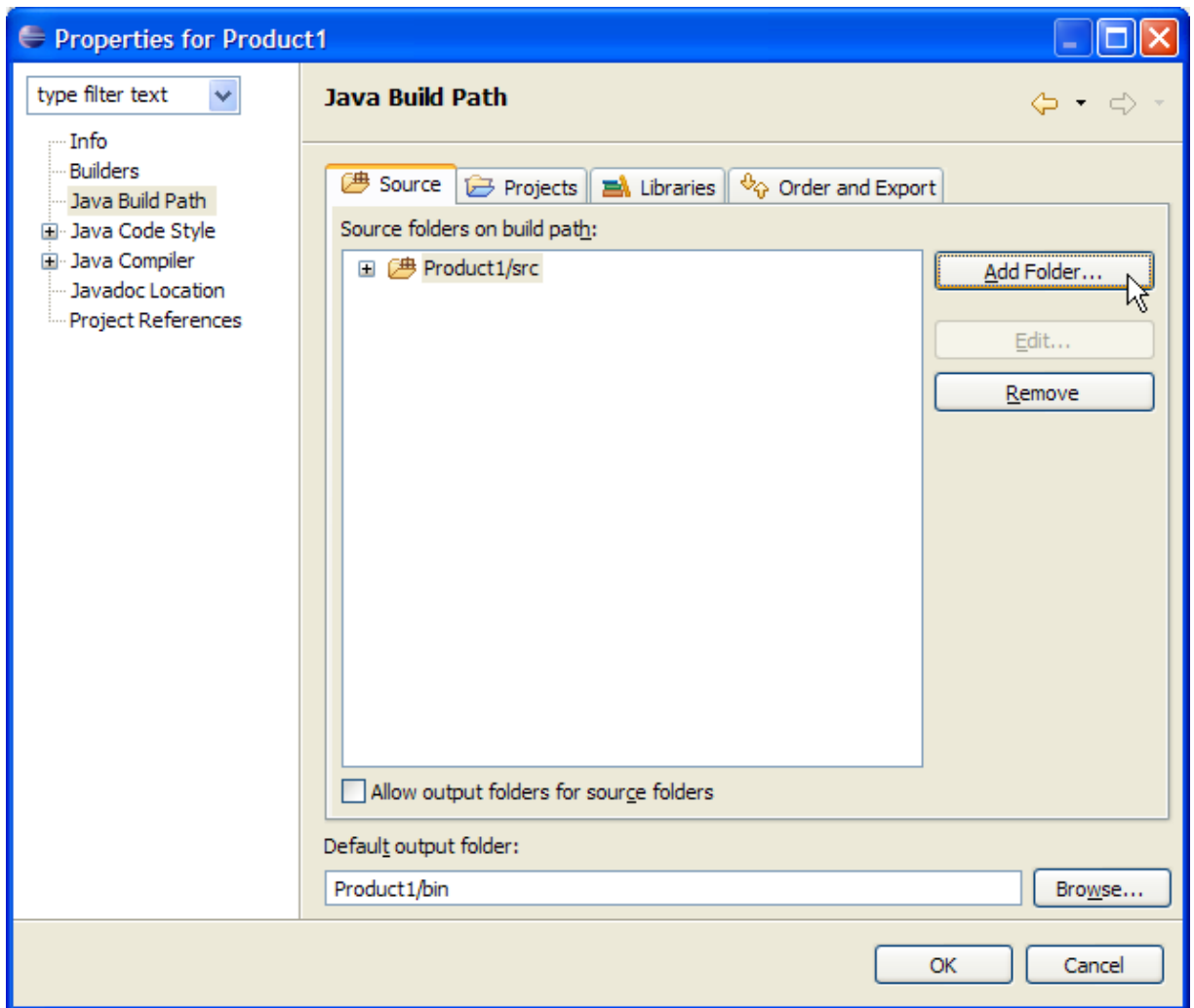


- Your project source setup now looks as follows:

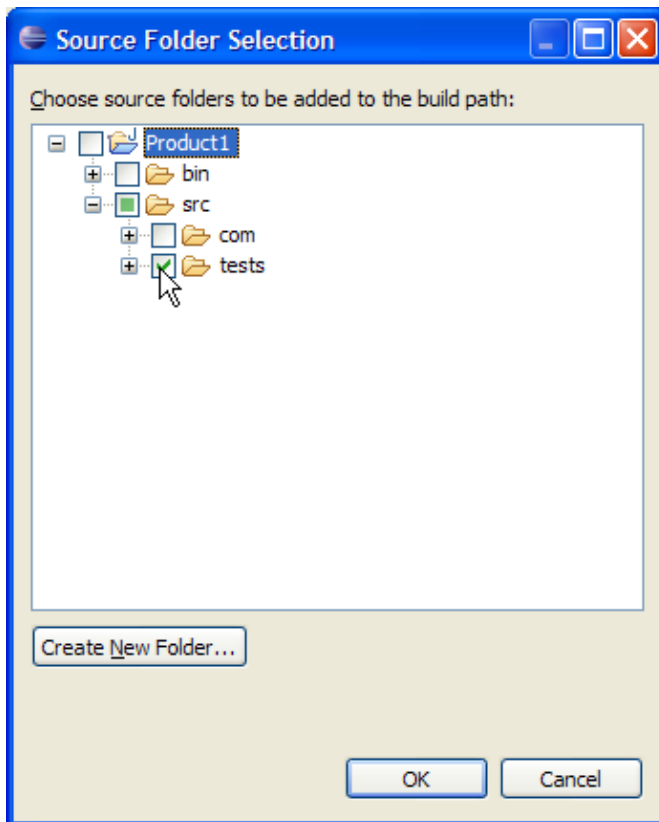


- Click **Finish**.
- Edit project "Product1" properties and select **Java Builder Path** page.

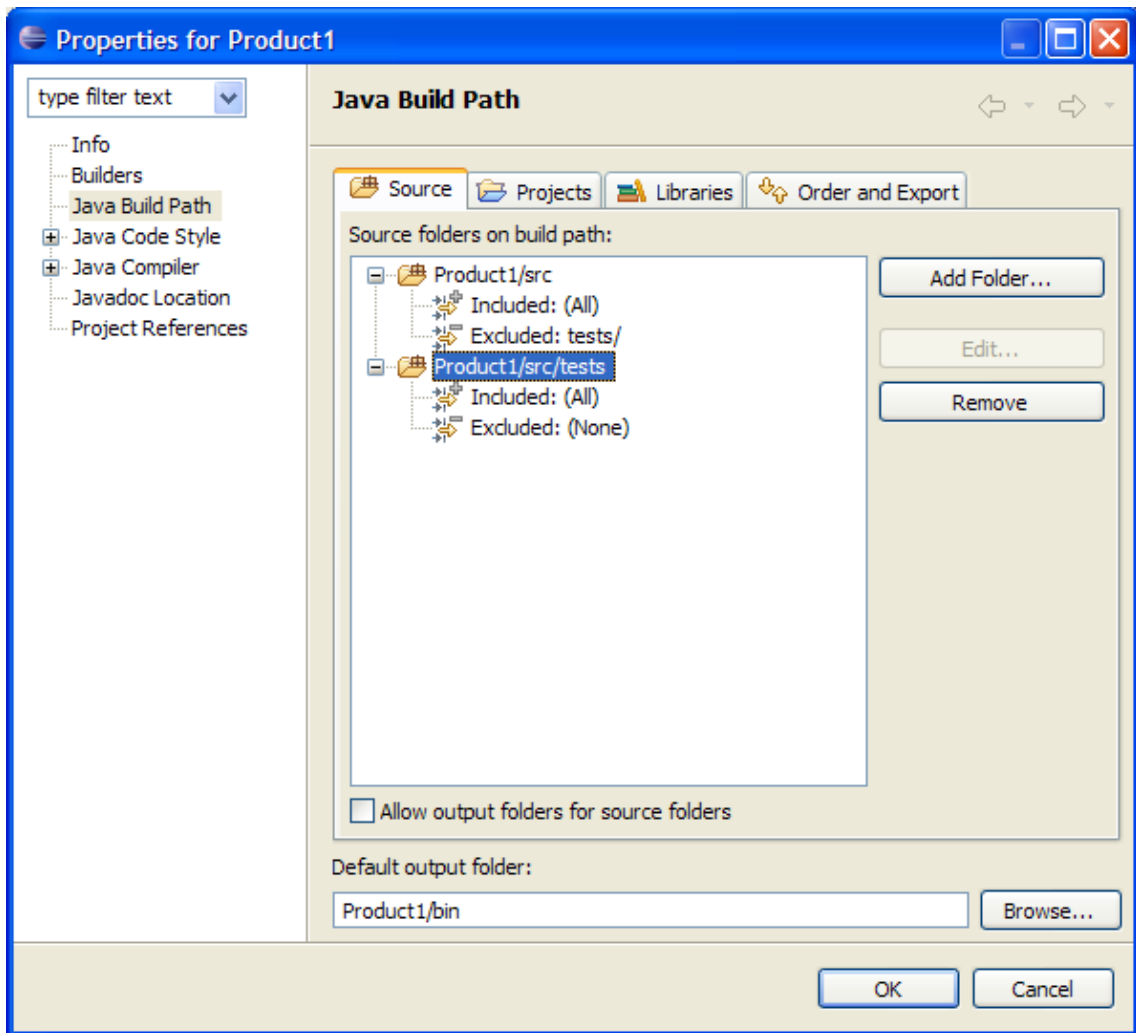
On **Source** tab, click **Add Folder...**



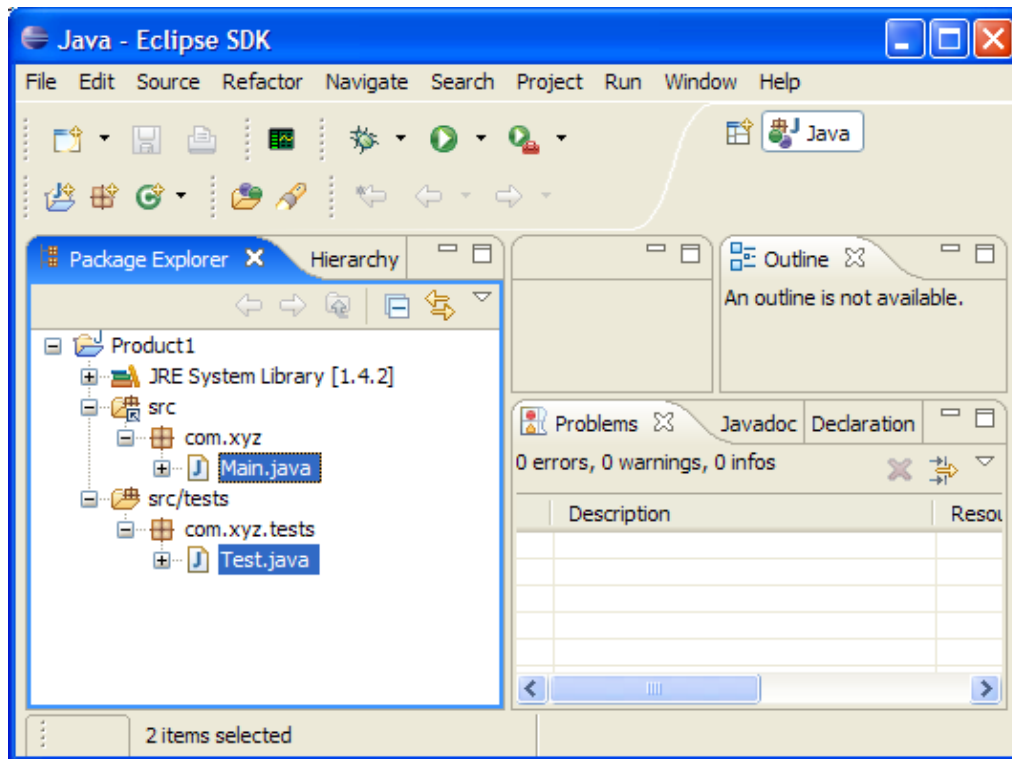
11. Expand "Product1", then "src" and select "tests".



12. Click **OK** to close the dialog. You get an information dialog saying that exclusion filters have been added. Click **OK**.
13. Your project setup now looks as follows:



14. Click **OK**.
15. You now have a Java project with a "src" folder and a "tests" folder which contain respectively the `D:\Product1\JavaSourceFiles` directory and the `D:\Product1\JavaSourceFiles\tests` directory.



■ Related concepts

[Java projects](#)

[Java views](#)

■ Related tasks

[Working with build paths](#)

[Creating a new Java project](#)

[Creating a Java project with source folders](#)

[Creating a new source folder](#)

[Using the Package Explorer](#)

■ Related reference

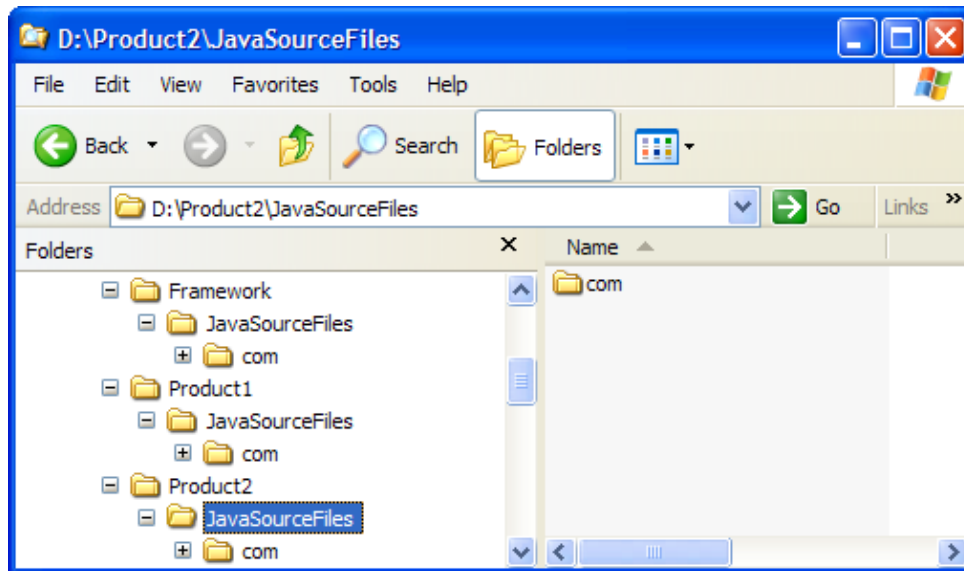
[New Java Project Wizard](#)

[Package Explorer View](#)

Products sharing a common source framework

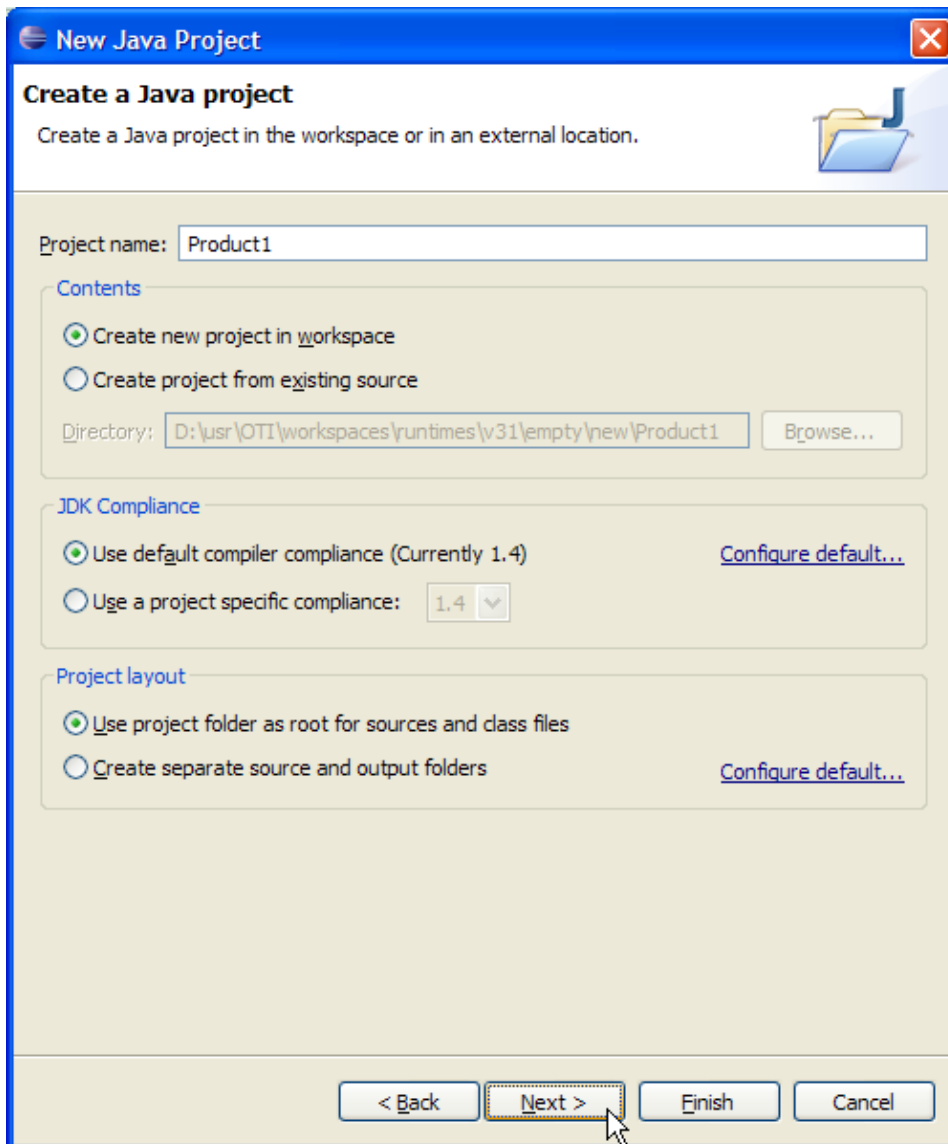
Layout on file system

- The Java source files for two products require a common framework.
- Projects and common framework are in separate directories which have their own source and output folders.




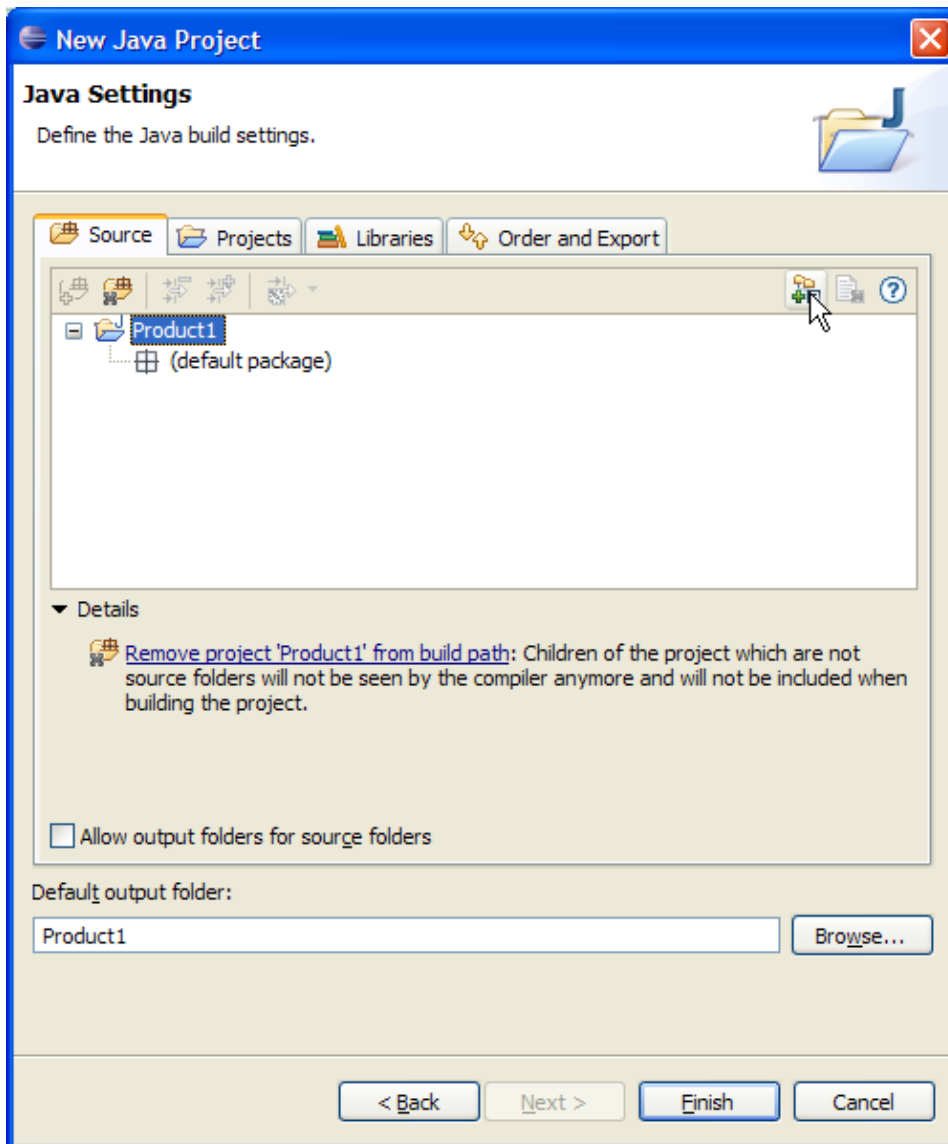
Steps for defining corresponding projects

1. Open a Java perspective, select the menu item **File > New > Project....** to open the **New Project** wizard.
2. Select **Java project** in the list of wizards and click **Next**.
3. On the next page, type "Product1" in the **Project name** field. Click **Next**.



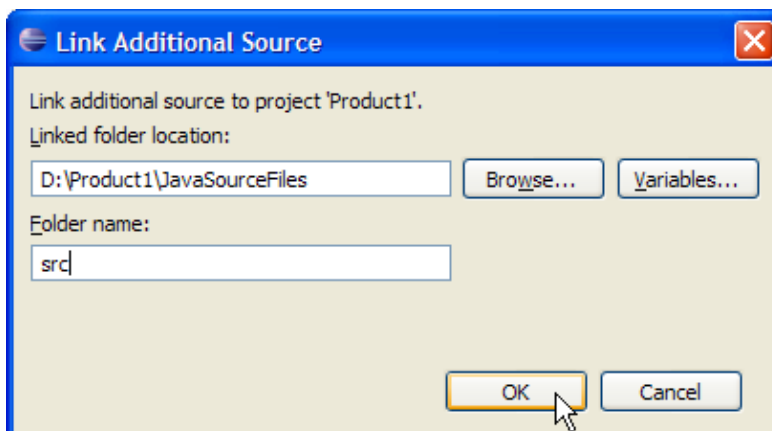
4. On the next page, Select "Product1" source folder.

Click **Link Additional Source to Project** button  in view bar.



5. In **Link Additional Source** click **Browse....** and choose the `D:\Product1\JavaSourceFiles` directory.

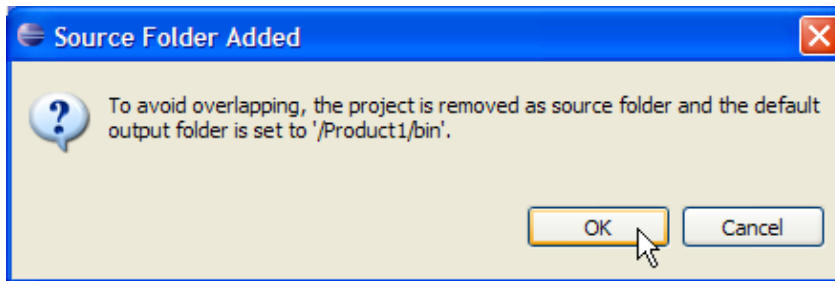
Type "src" in the **Folder name** field.



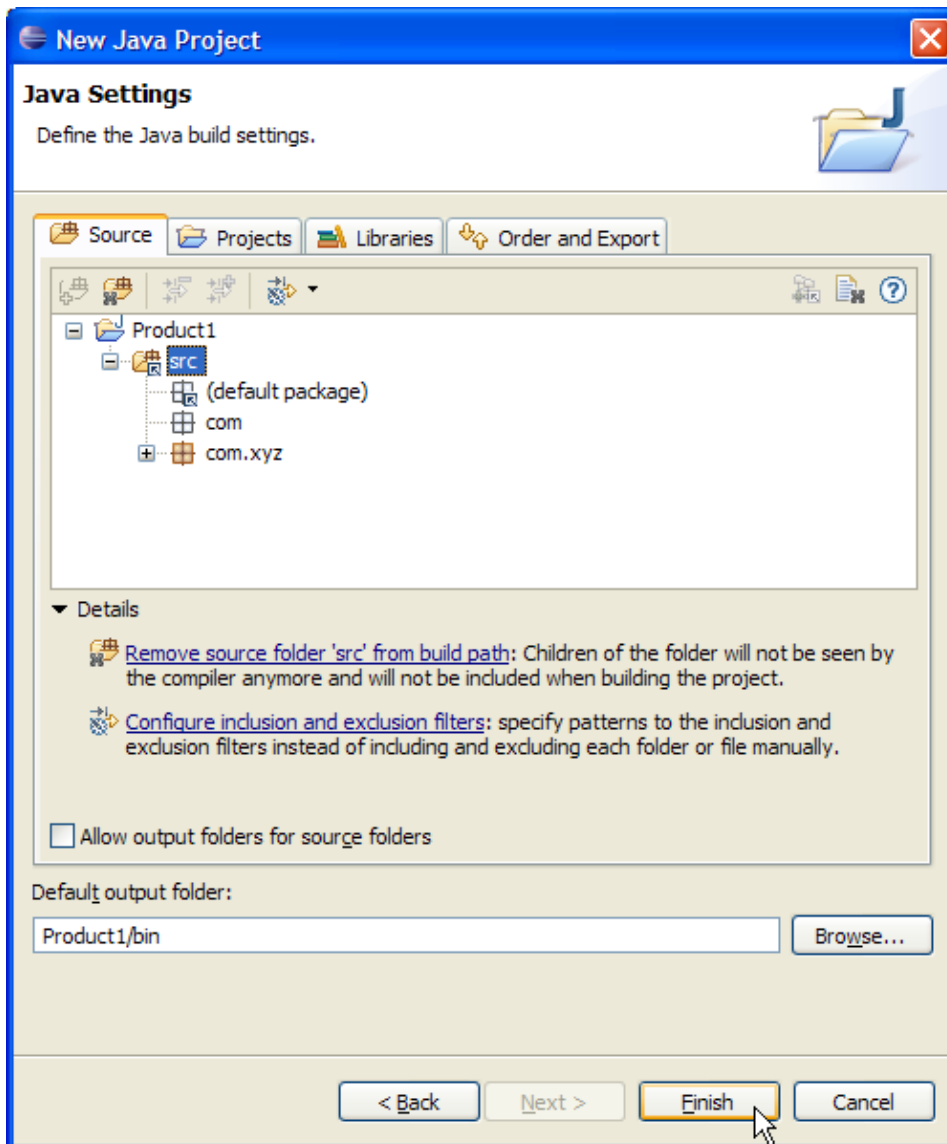
6. Click **OK** to close the dialog.

Basic tutorial

- Click **OK** in confirmation dialog to have "Product1/bin" as default output folder.

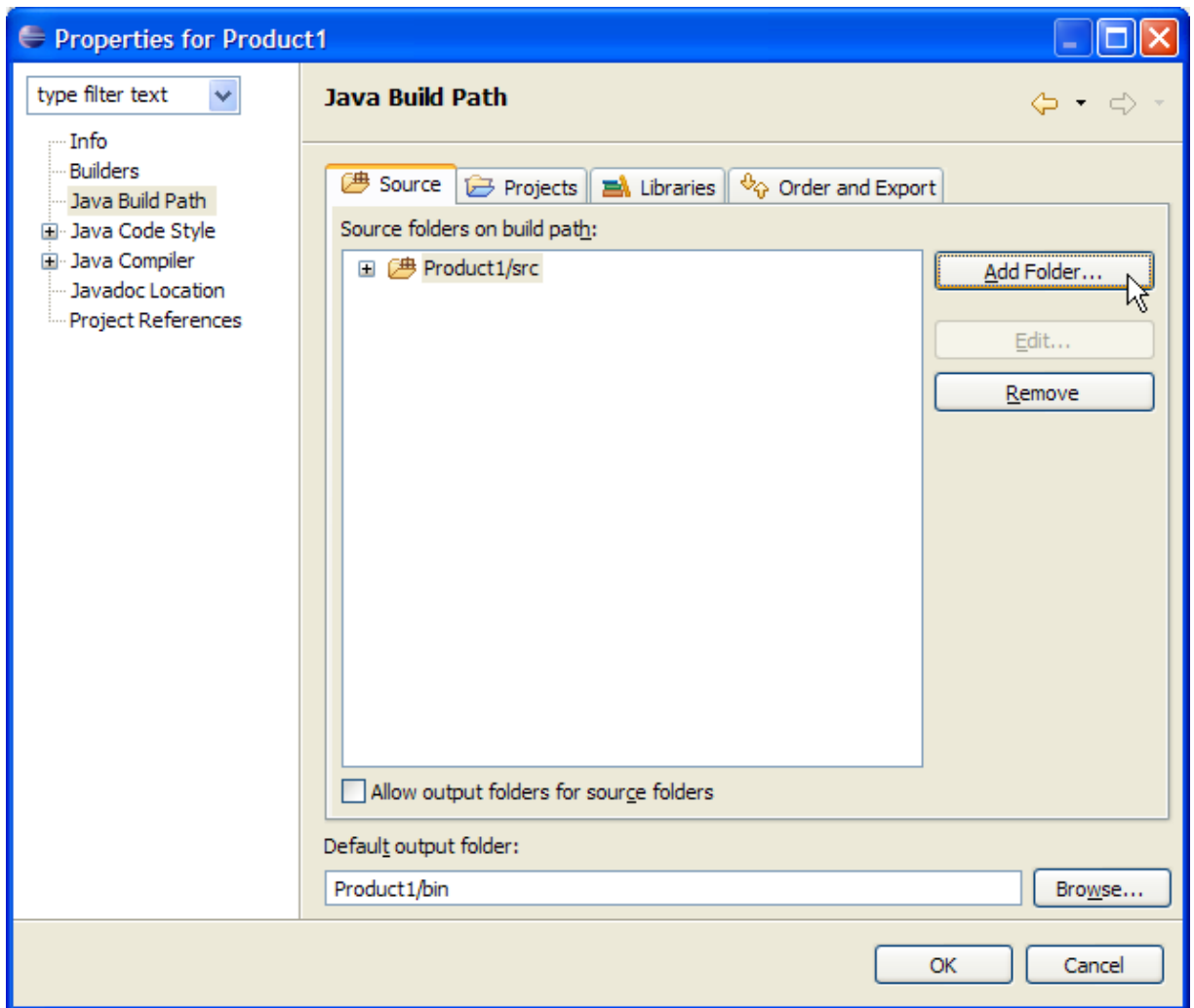


- Your project source setup now looks as follows:

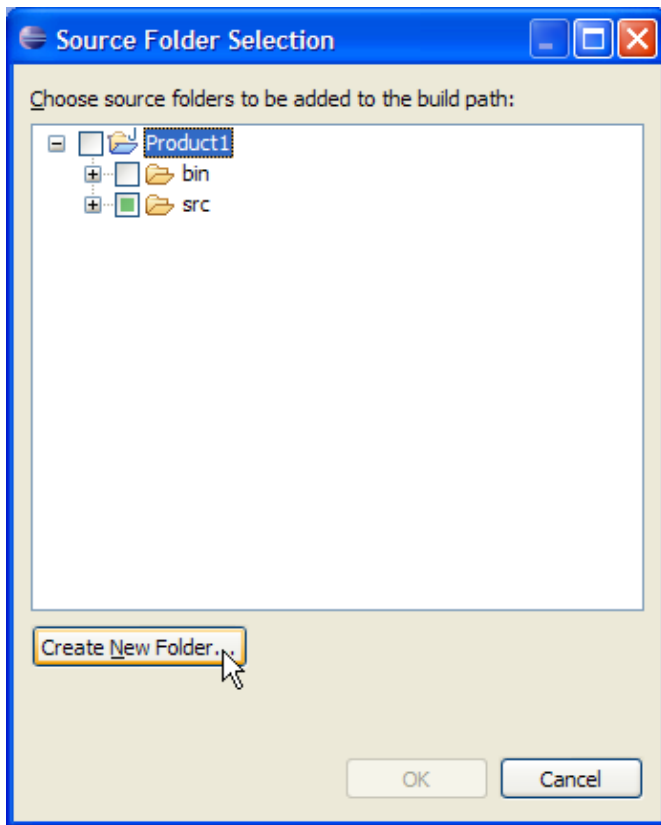


- Click **Finish**.
- Edit project "Product1" properties and select **Java Builder Path** page.

On **Source** tab, click **Add Folder...**



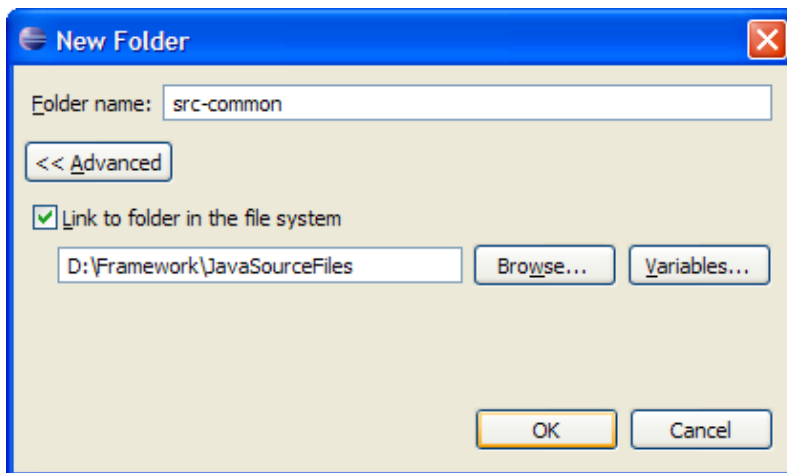
11. On *Source Folder Selection* click *Create New Folder...*



12. In **New Folder**, type "src-common" in the **Folder name** field.

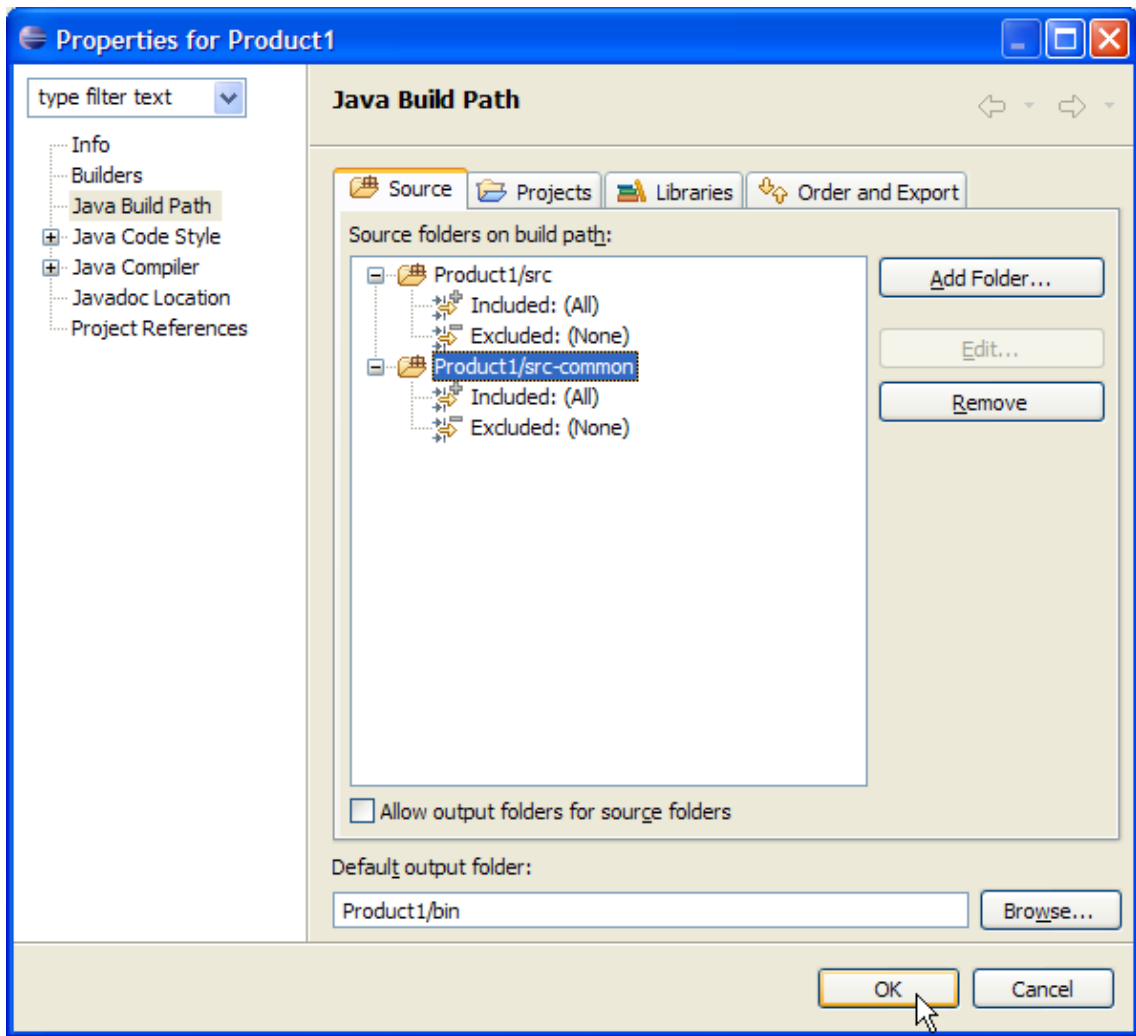
Click **Advanced>>** and check **Link to folder in the file system**.

Then click **Browse....** and choose the `D:\Framework\JavaSourceFiles` directory.

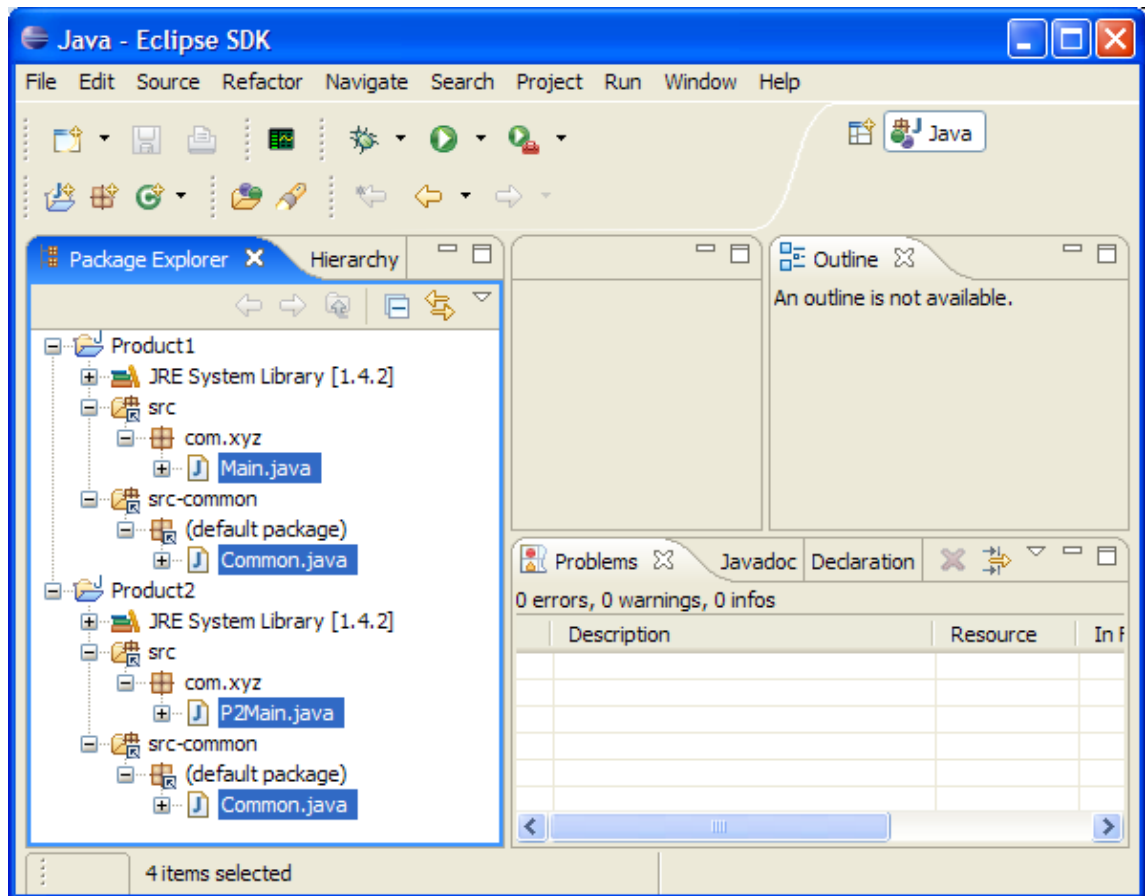


13. Click **OK** twice to close dialogs.

14. Your project setup now looks as follows:



15. Click **OK**.
16. Repeat these steps for "Product2".
17. You now have two Java projects which respectively contain the sources of "Product1" and "Product2" and which are using the sources of "Framework".



Note: Files in "src-common" are shared. So editing "Common.java" in "Product1" will modify "Common.java" in "Product2". However they are compiled in the context of their respective projects. Two "Common.class" files will be generated; one for each project. If the two projects have different compiler options, then different errors could be reported on each "Common.java" file.

■ Related concepts

[Java projects](#)

[Java views](#)

■ Related tasks

[Working with build paths](#)

[Creating a new Java project](#)

[Creating a Java project with source folders](#)

[Creating a new source folder](#)

[Using the Package Explorer](#)

■ Related reference

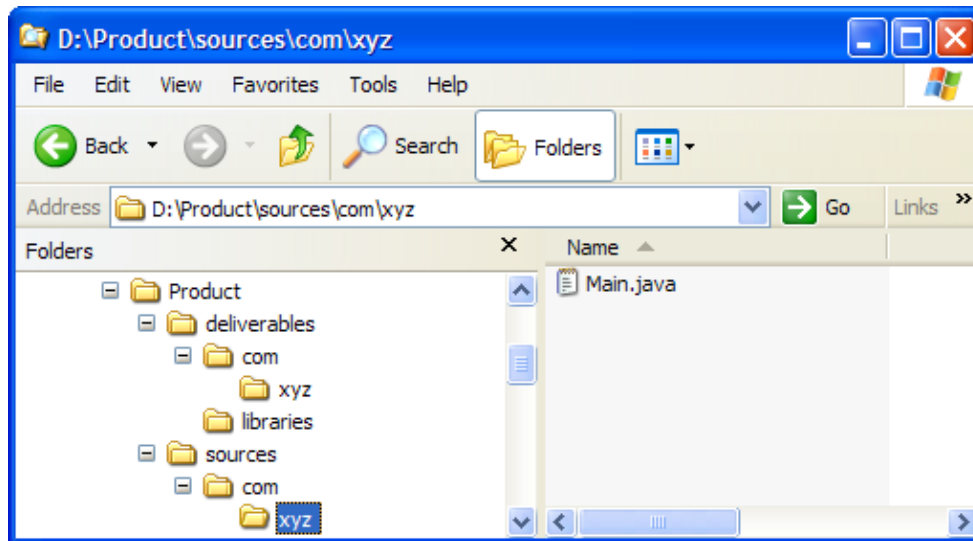
[New Java Project Wizard](#)

[Package Explorer View](#)

Nesting resources in output directory

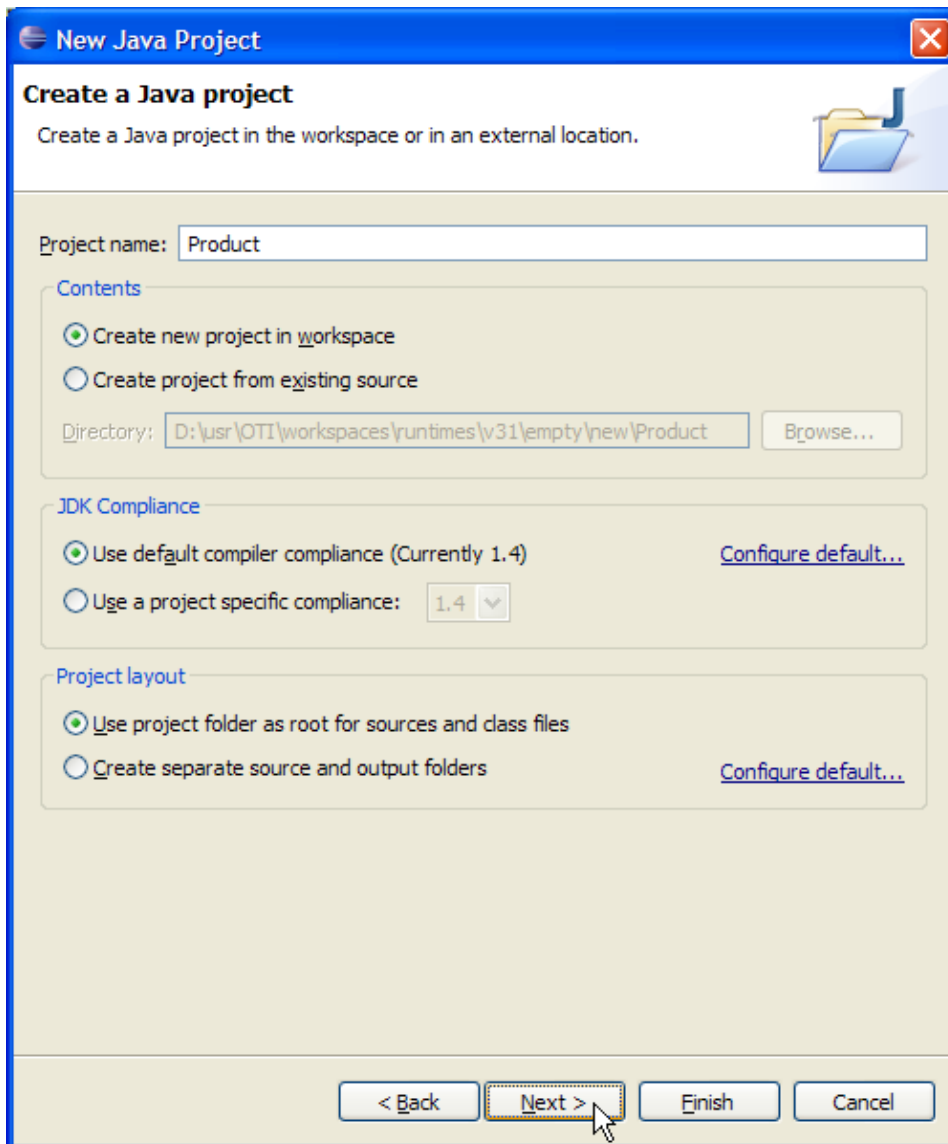
Layout on file system

- The Java source files for a product are laid out both in *sources* and *deliverables* directories.
- All Java class files are laid out in *deliverables* directory.
- Project needs to use some libraries located in *deliverables/libraries* directory:




Steps for defining a corresponding project

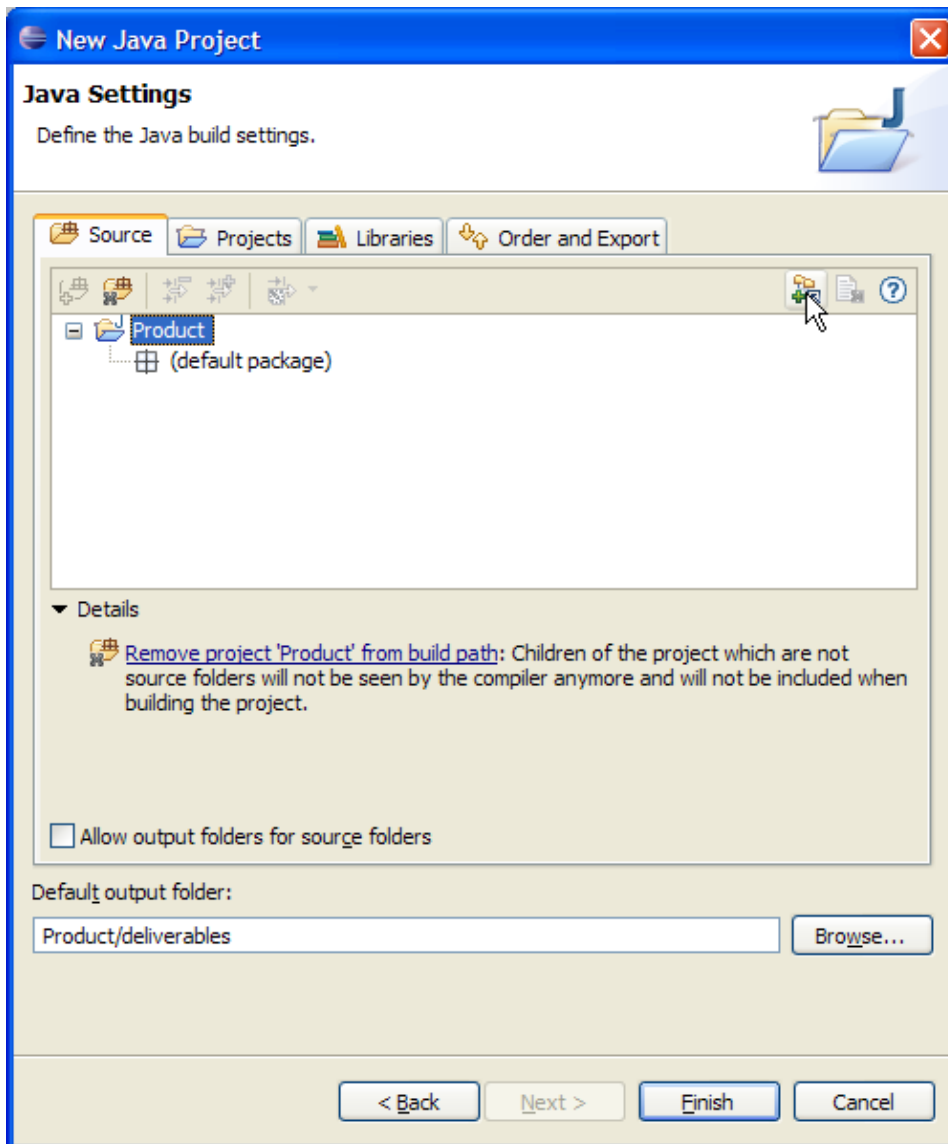
1. Open a Java perspective, select the menu item **File > New > Project....** to open the *New Project* wizard.
2. Select **Java project** in the list of wizards and click *Next*.
3. On the next page, type "Product" in the *Project name* field. Click *Next*.



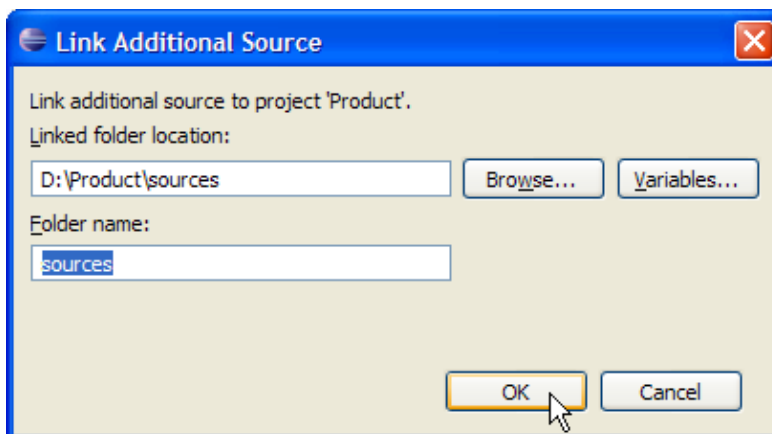
4. On the next page, Type "Product/deliverables" in **Default output folder** field.

Select "Product1" source folder.

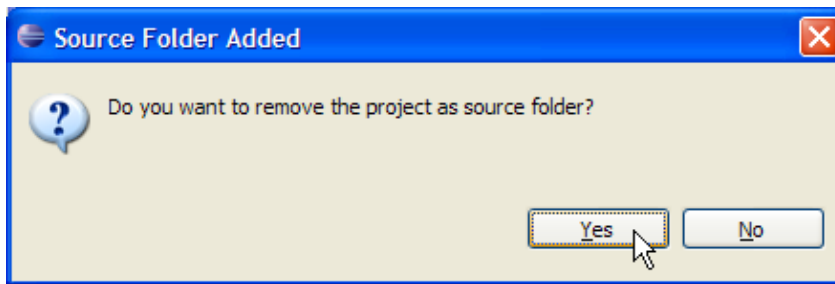
Click **Link Additional Source to Project** button  in view bar.



5. In **Link Additional Source** click **Browse....** and choose the `D:\Product\sources` directory.

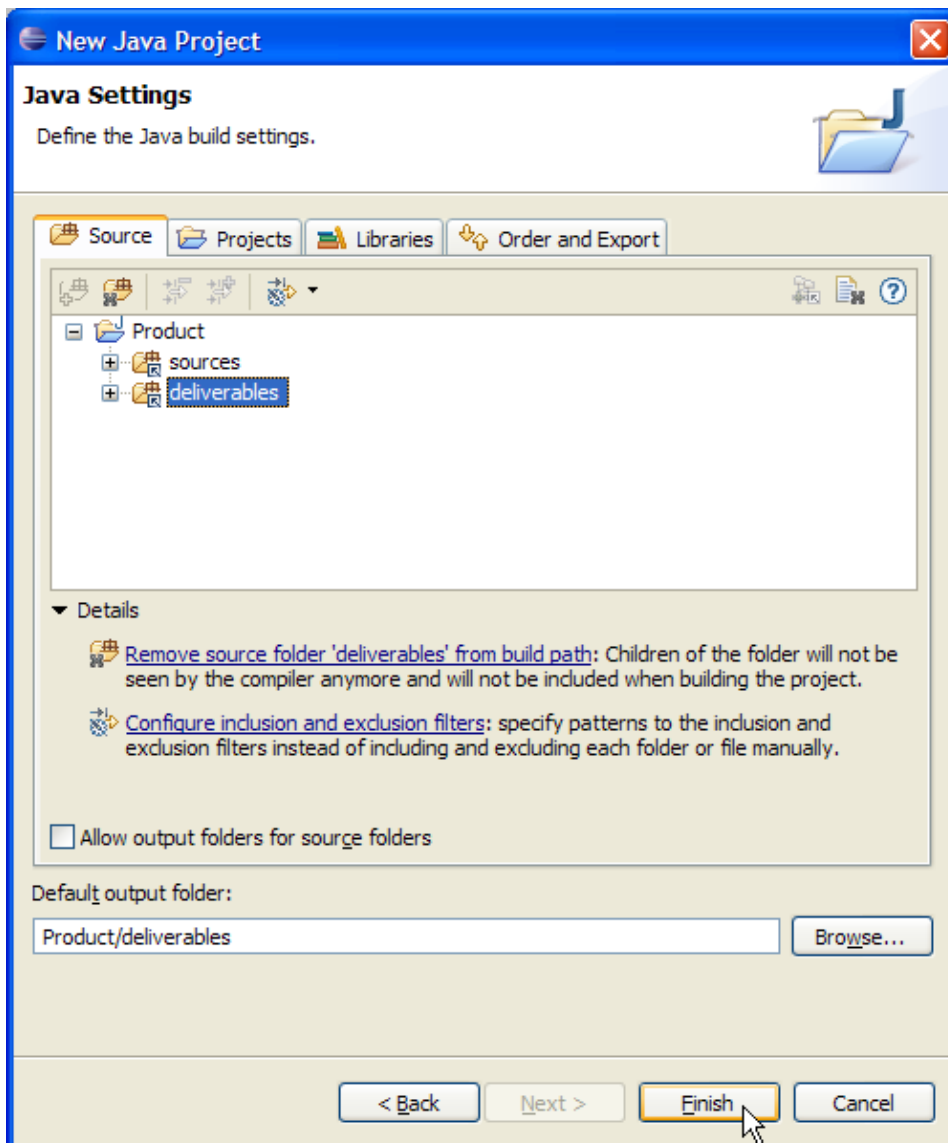


6. Click **OK** to close the dialog.
 7. Click **OK** in confirmation dialog to remove the project as source folder.



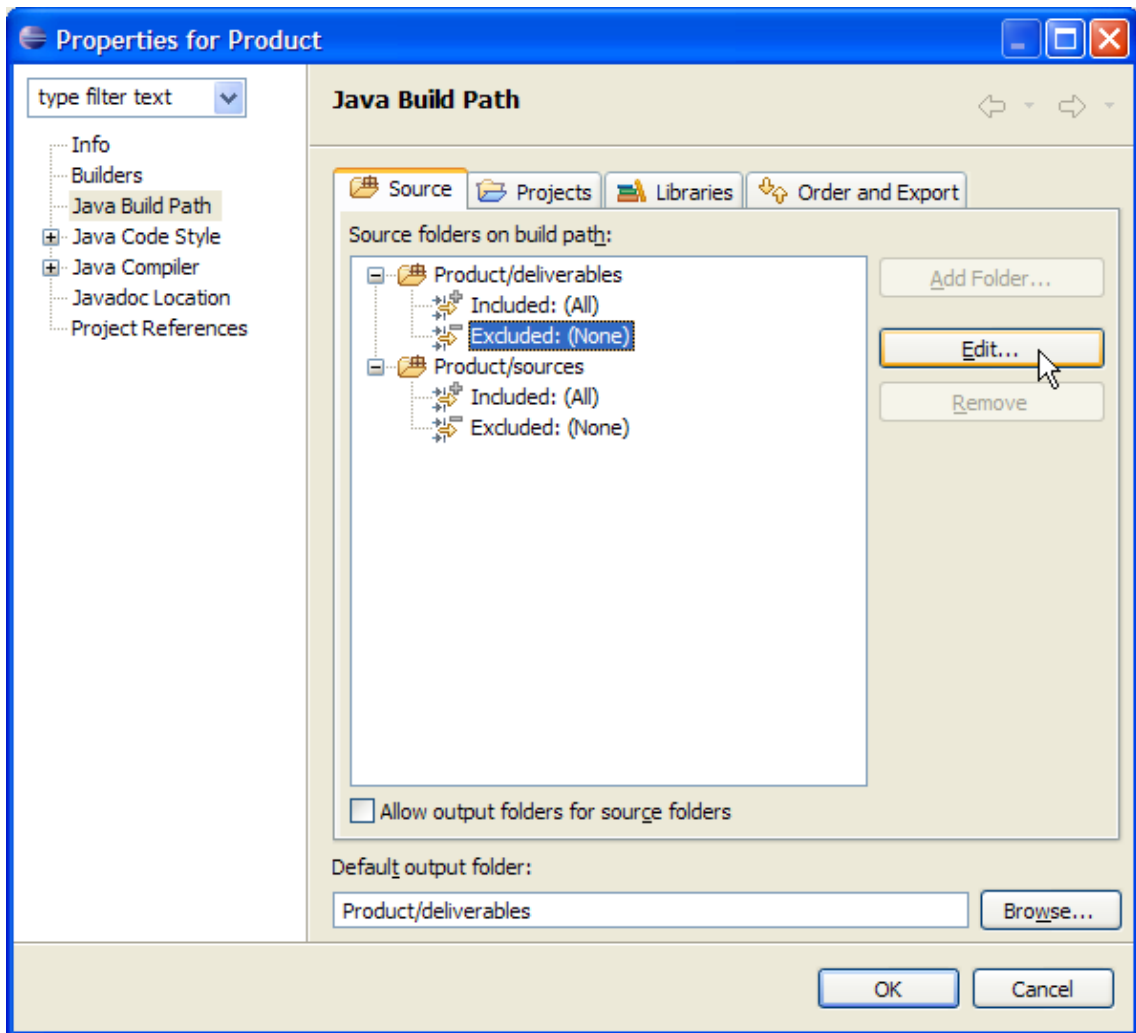
8. Repeat previous steps to create source folder "deliverables" linked to `D:\Product\deliverables` directory.

Click **Finish**.

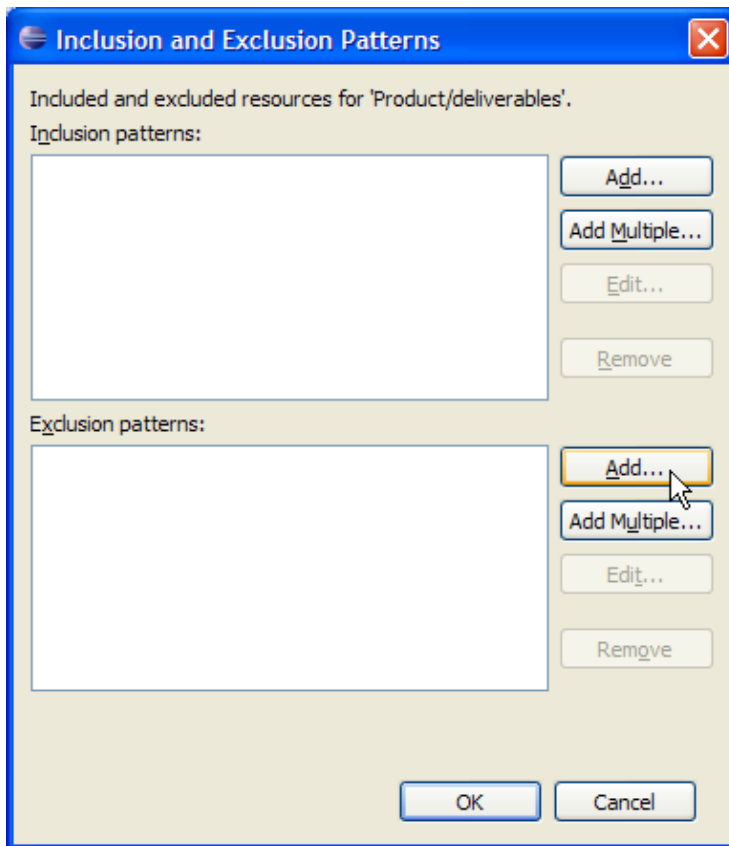


9. Edit project "Product1" properties and select **Java Builder Path** page.

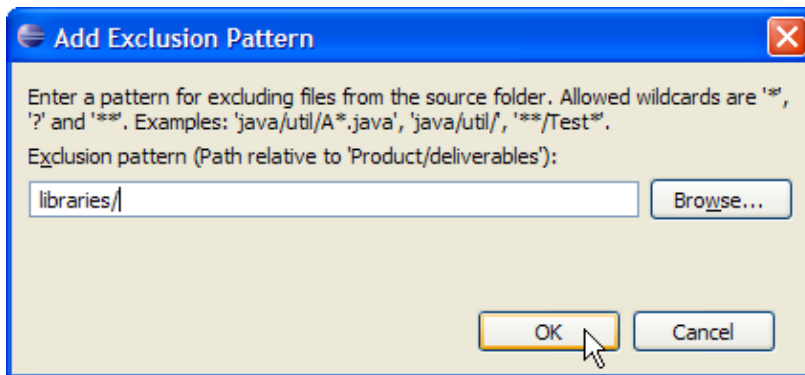
Expand "Product/deliverables", select **Excluded** and click **Edit...**



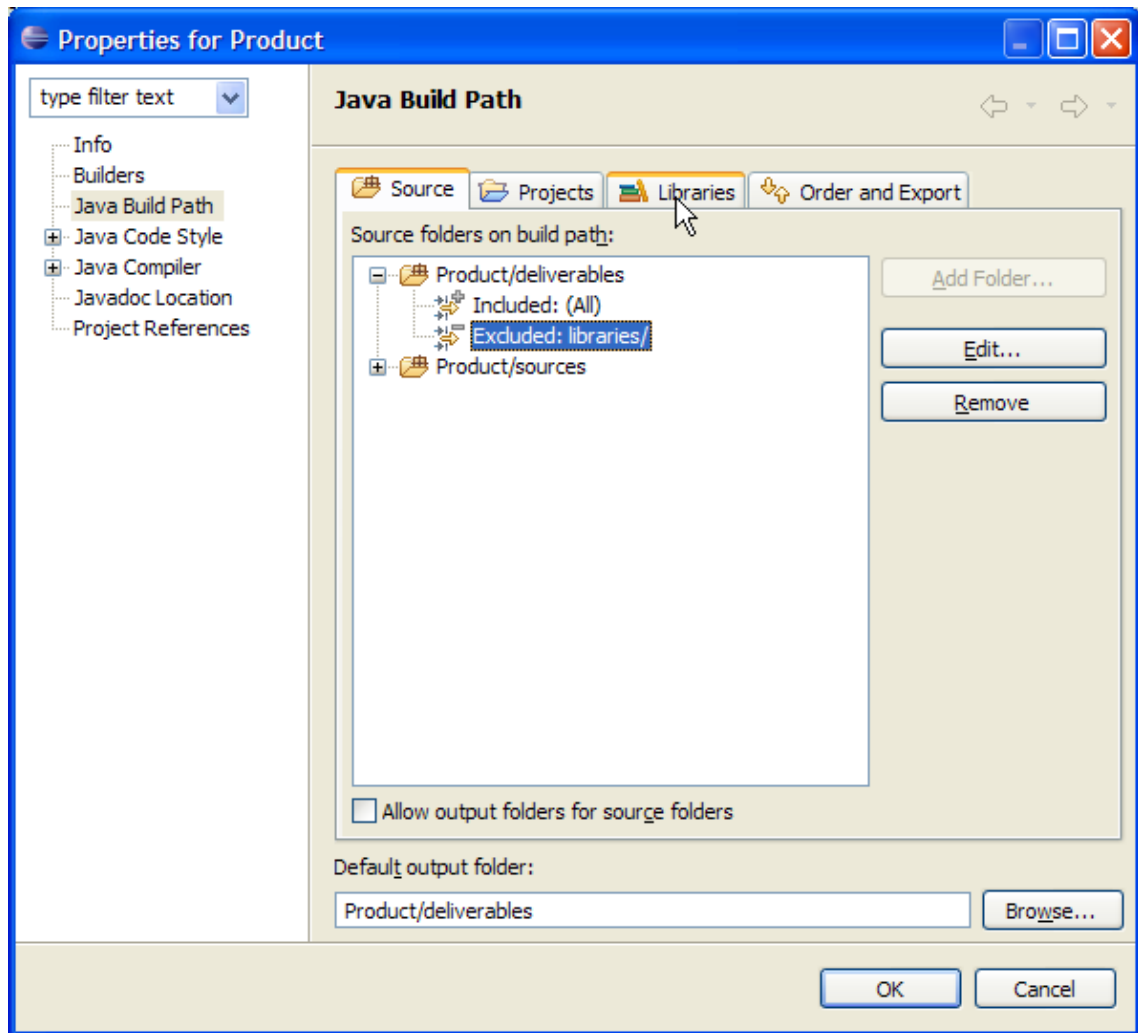
10. Click **Add...** in *Exclusion patterns* part of the *Inclusion and Exclusion Patterns* dialog.



11. Type "libraries/" in *Add Exclusion Pattern* dialog and click **OK** to validate and close the dialog.

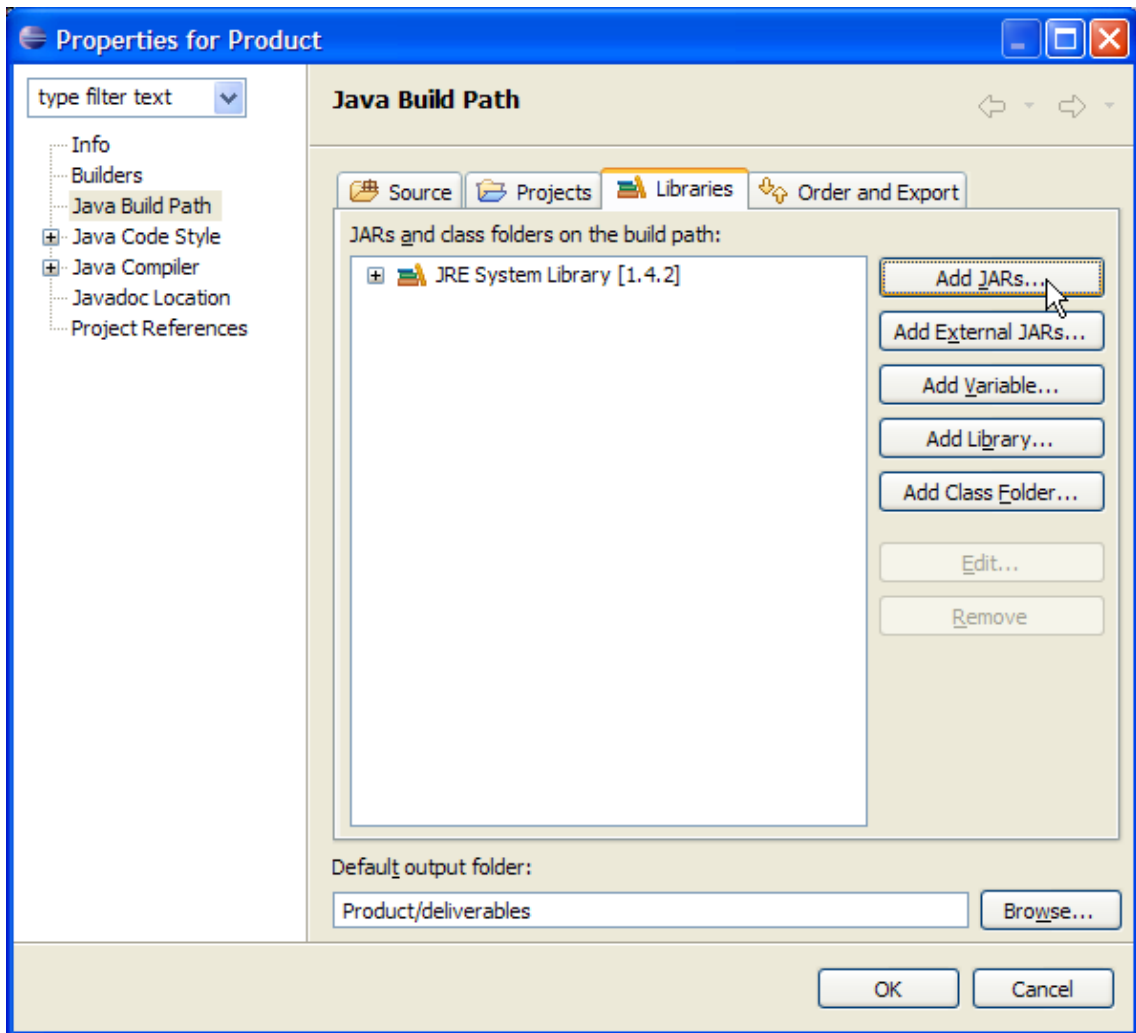


12. Click **OK** twice to close dialogs.
13. Your project source setup now looks as follows:



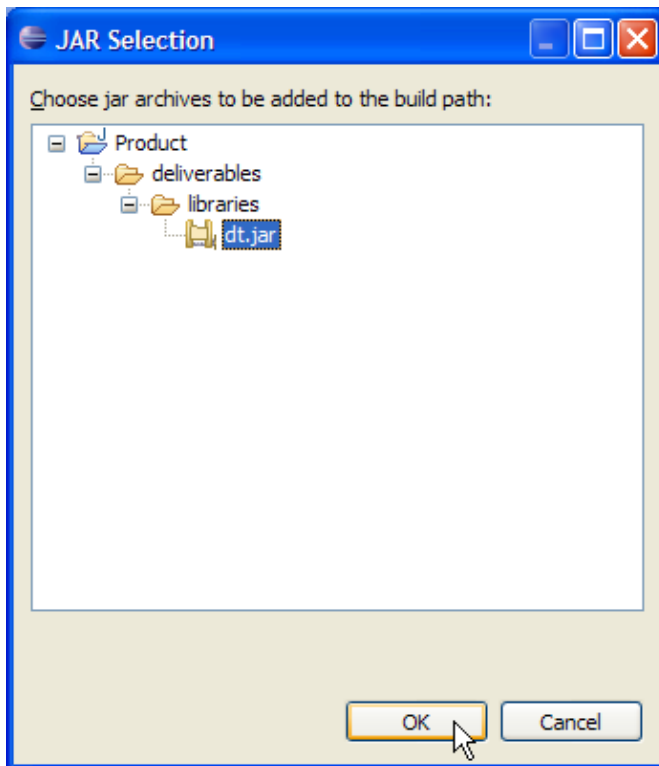
14. Select **Libraries** tab.

Click on **Add JARs...**

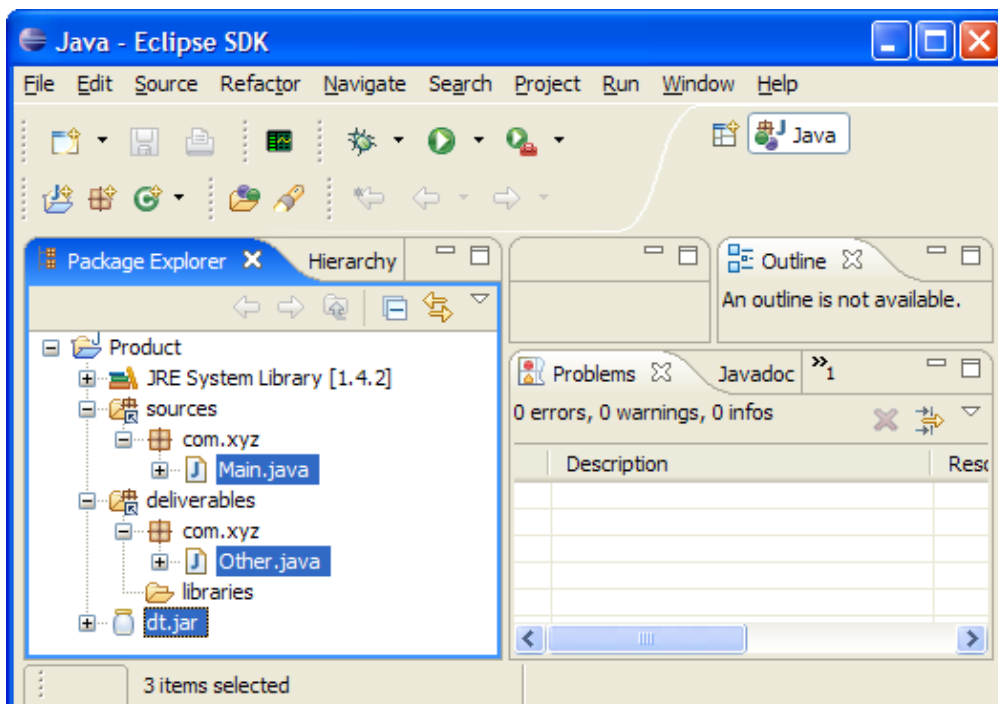


15. Expand "Product" hierarchy to select jar files in "libraries" directory

Click **OK**.



16. You now have a Java project with a "sources" folder and an output folder which contains nested library resources.



● Related concepts

[Java projects](#)

[Java views](#)

■ Related tasks

[Working with build paths](#)

[Creating a new Java project](#)

[Creating a Java project with source folders](#)

[Creating a new source folder](#)

[Using the Package Explorer](#)

■ Related reference

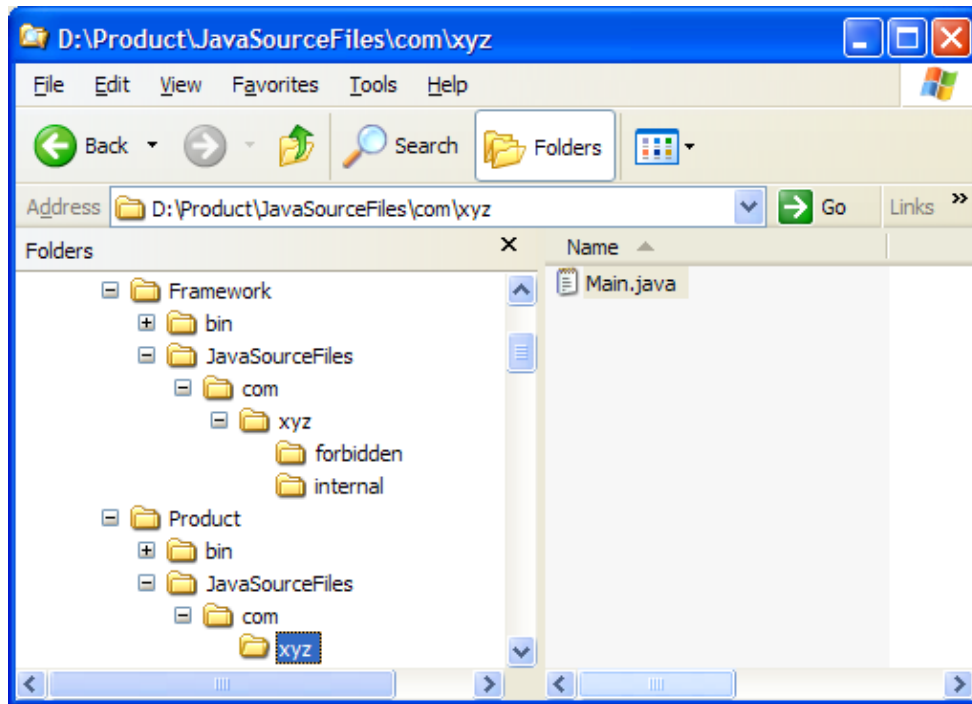
[New Java Project Wizard](#)

[Package Explorer View](#)

Project using a source framework with restricted access

Layout on file system

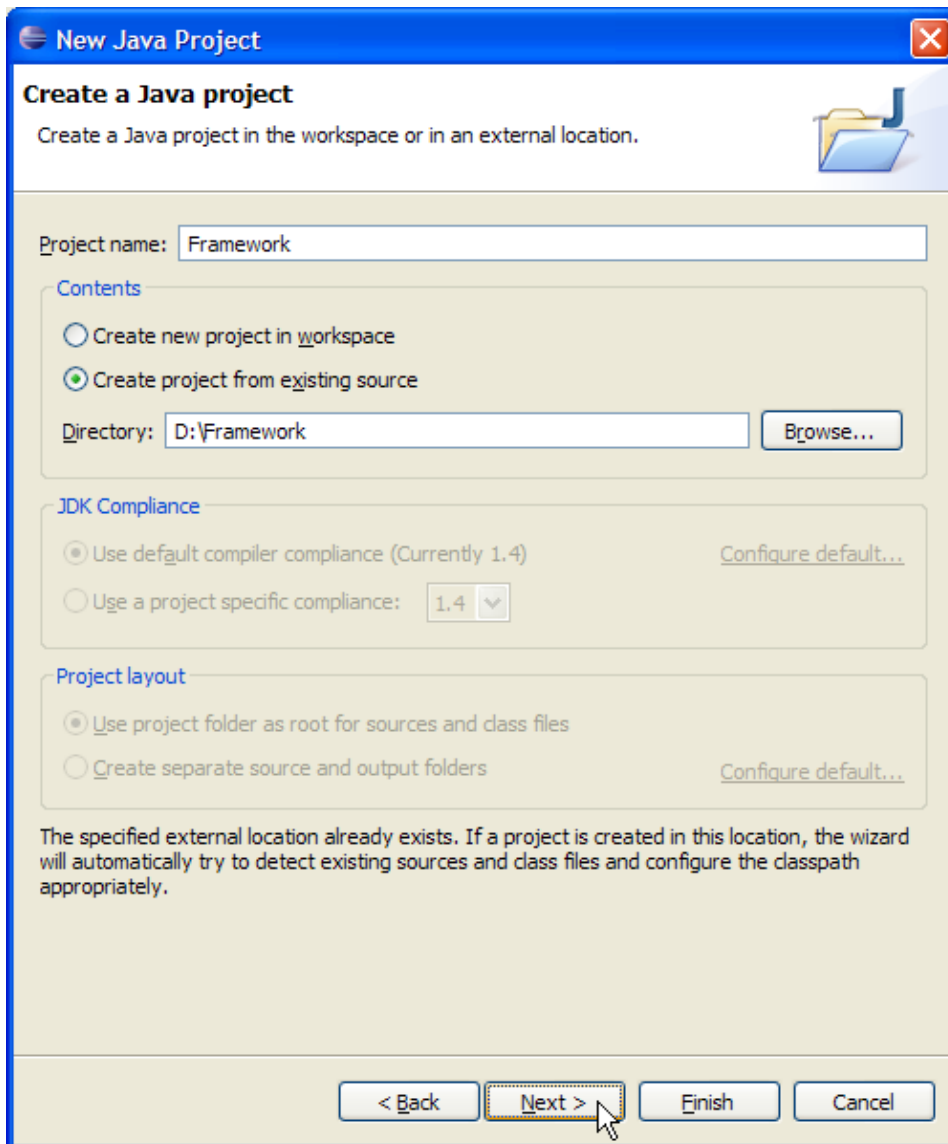
- The Java source files for a product requires a source framework.
- "Product" and "Framework" are in separate directories which have their own source and output folders.



Steps for defining corresponding projects

1. Open a Java perspective, select the menu item **File > New > Project...** to open the *New Project* wizard.
2. Select **Java project** in the list of wizards and click *Next*.
3. On the next page, type "Framework" in the **Project name** field.
4. In **Contents** group, change selection to *Create project from existing source*.

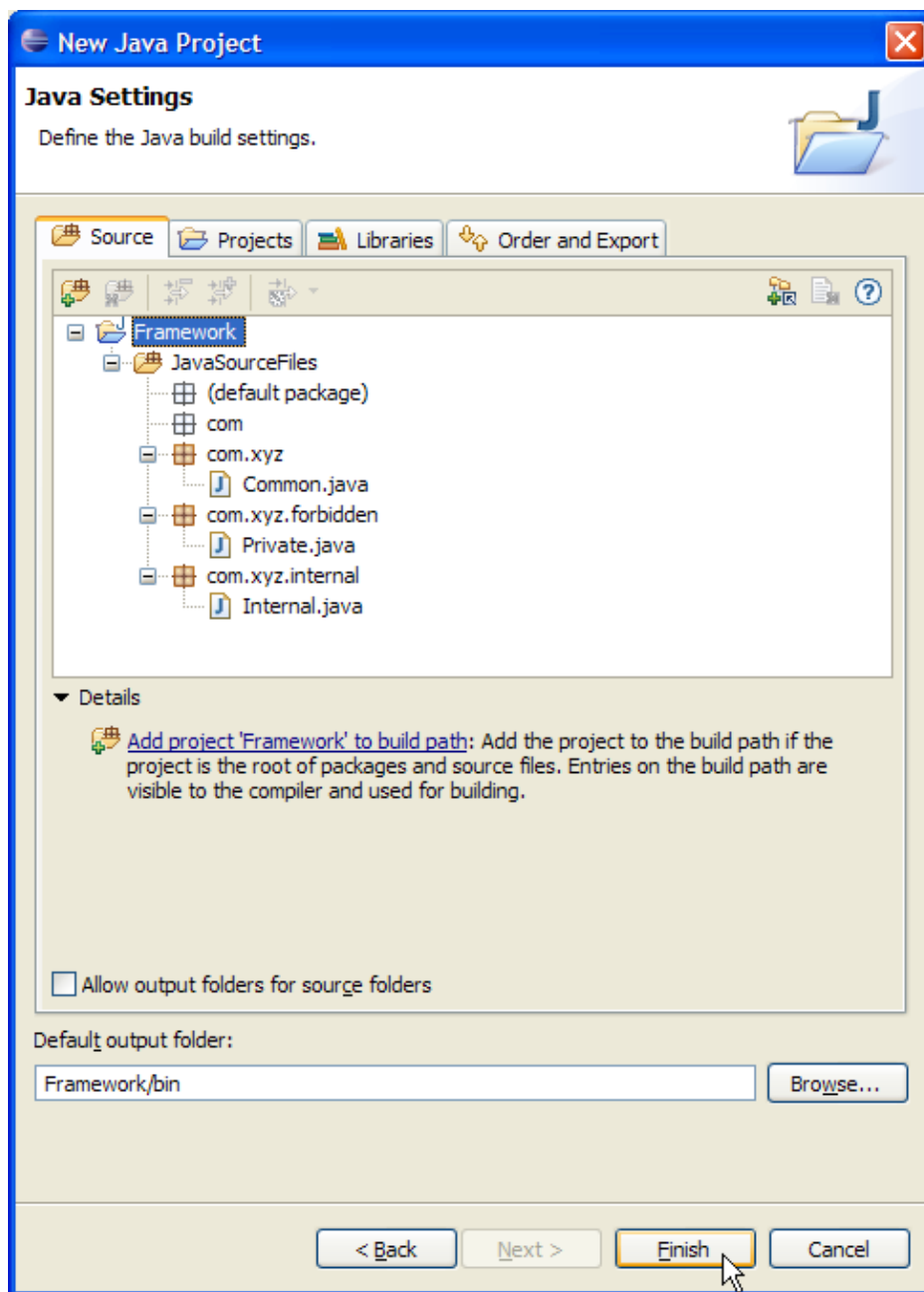
Click **Browse...** and choose the `D:\Framework` directory.



Click *Next*.

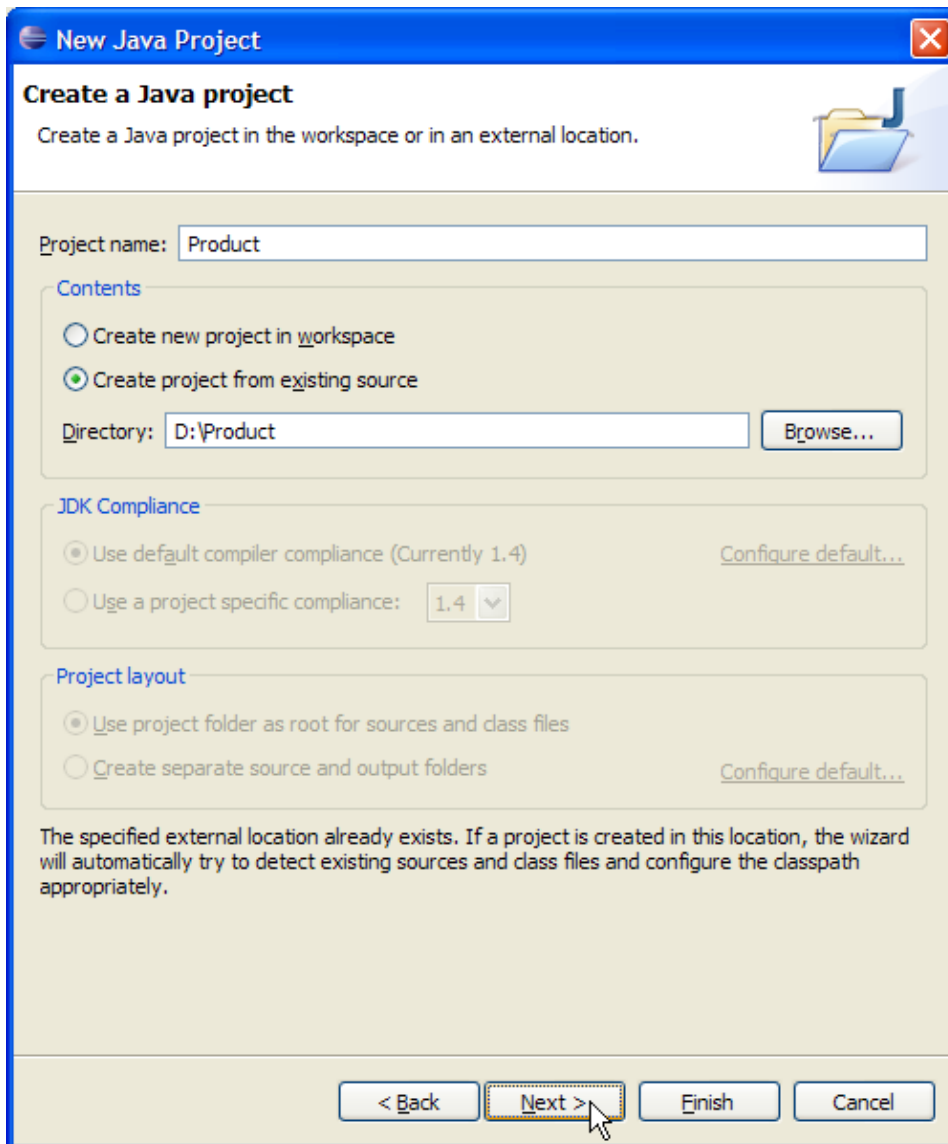
5. On the next page, verify that directory *JavaSourceFiles* has been automatically added as source folder.

Expand it to preview your project source folder contents:



6. Click **Finish**.
7. In Java perspective, type **Ctrl+N** to open **New** wizards dialog.
- Select **Java project** in the list of wizards and click **Next**.
8. On the next page, type "Product" in the **Project name** field.
9. In **Contents** group, change selection to **Create project from existing source**.

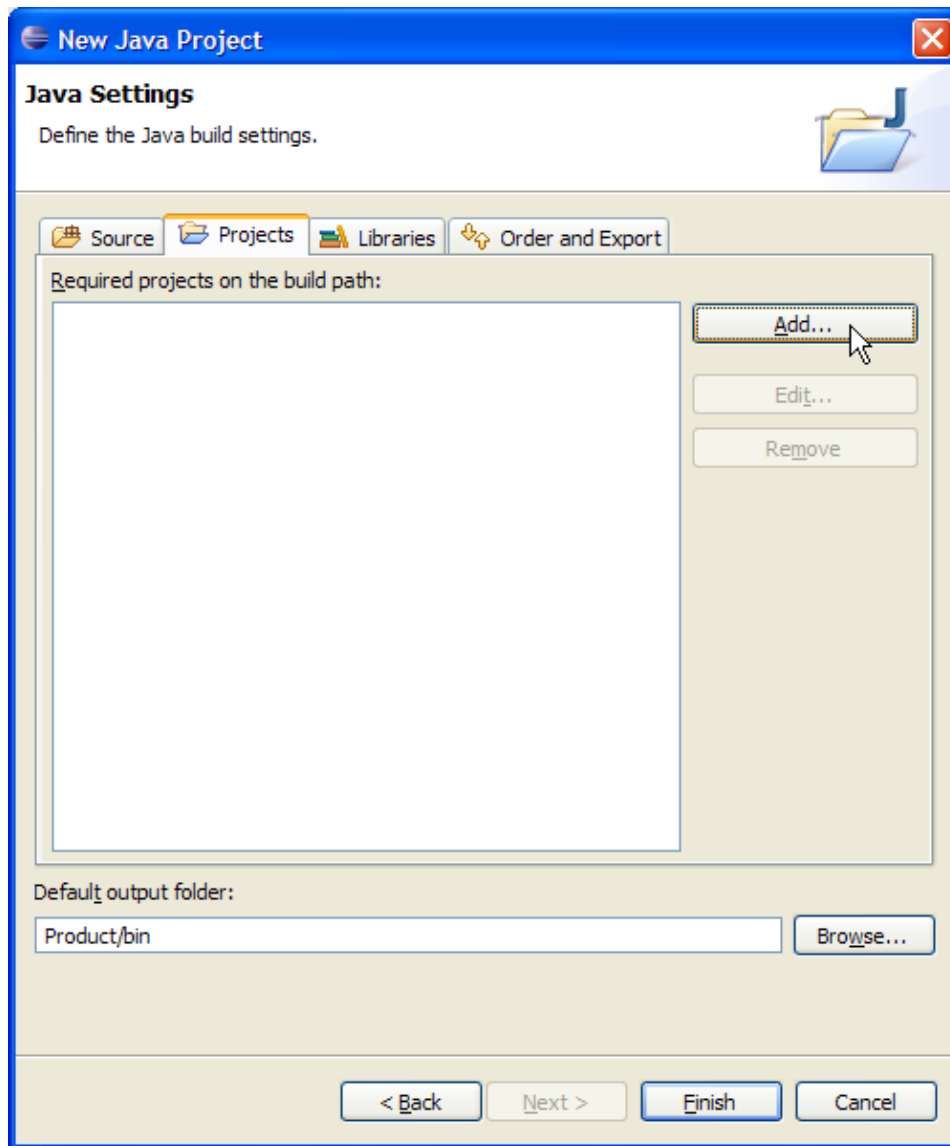
Click **Browse...** and choose the `D:\Product` directory.



Click *Next*.

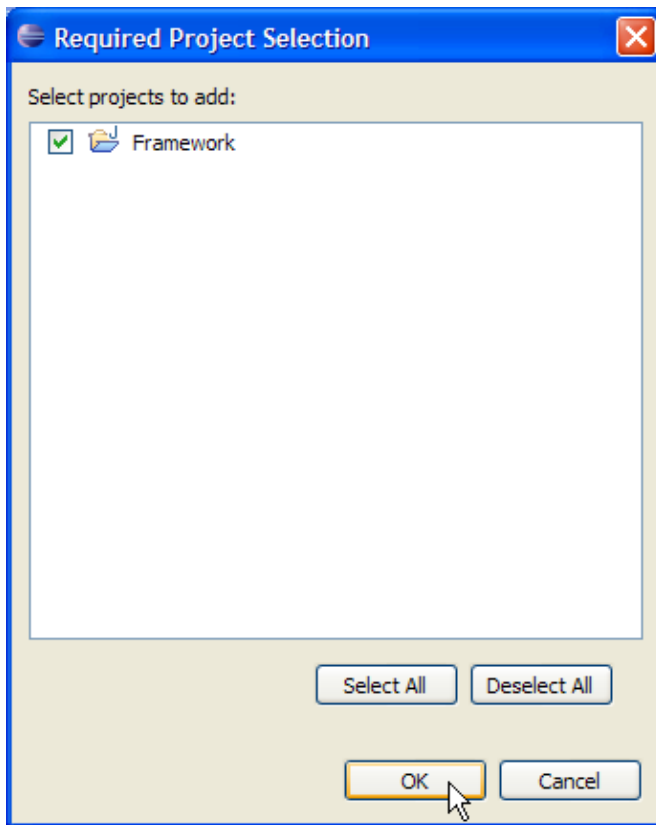
10. Let's add a dependency to source framework project...
11. On the next page, verify that directory *JavaSourceFiles* has been automatically added as source folder.

Select *Projects* tab.



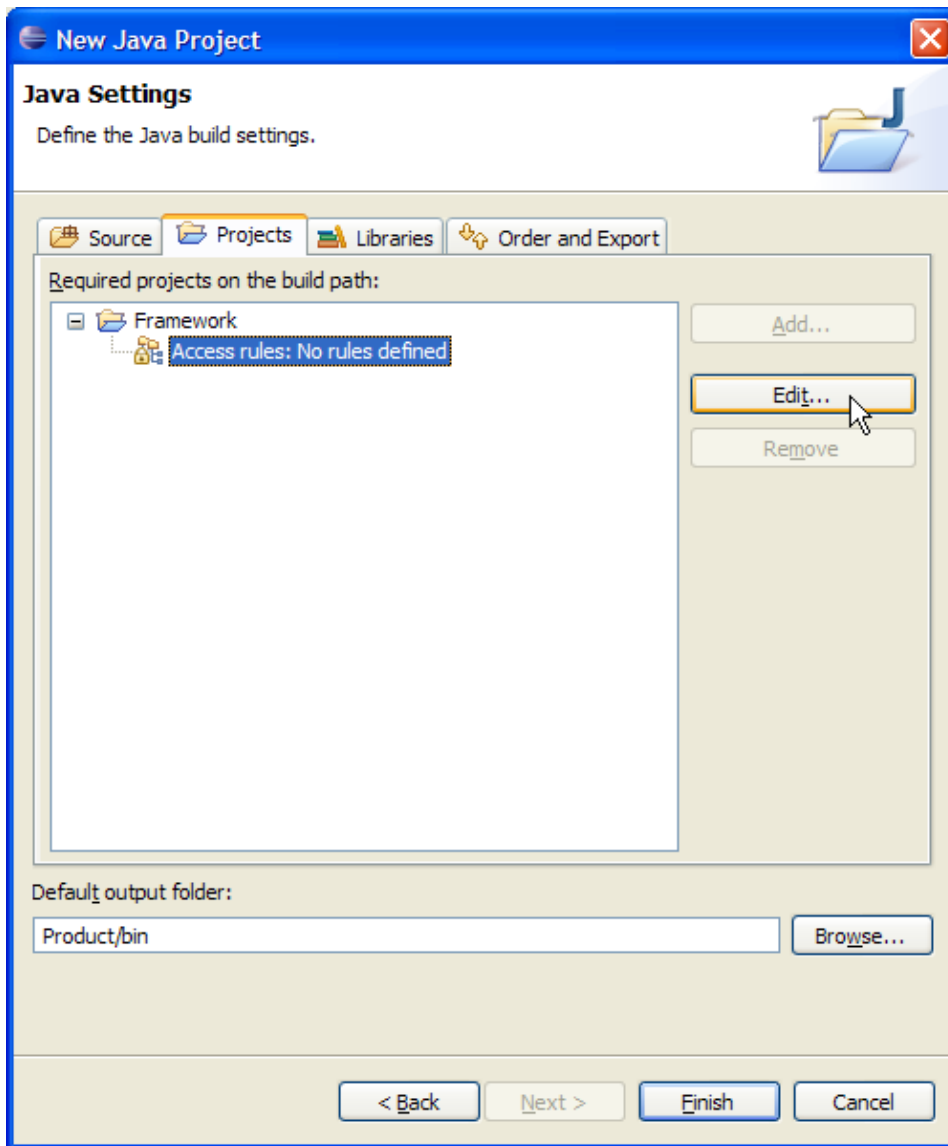
Click **Add...**

12. In **Required Project Selection**, check "Framework".



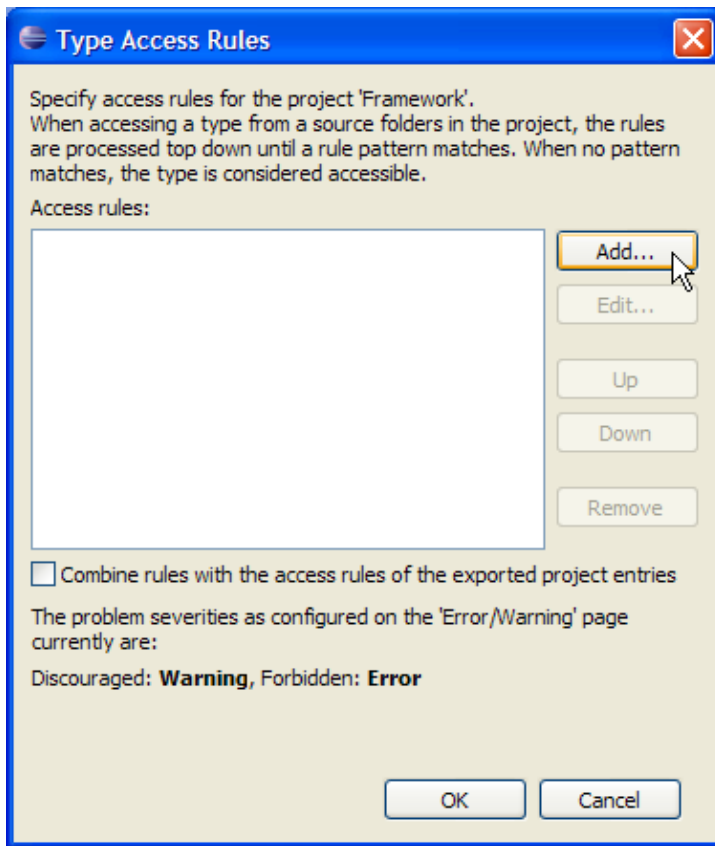
Click **OK**.

13. Now, let's put access rules on source framework content to authorize, discourage or forbid access to "Framework" source folders, package and classes...
14. In **Projects** tab, select "Access rules" of "Framework" depending project.



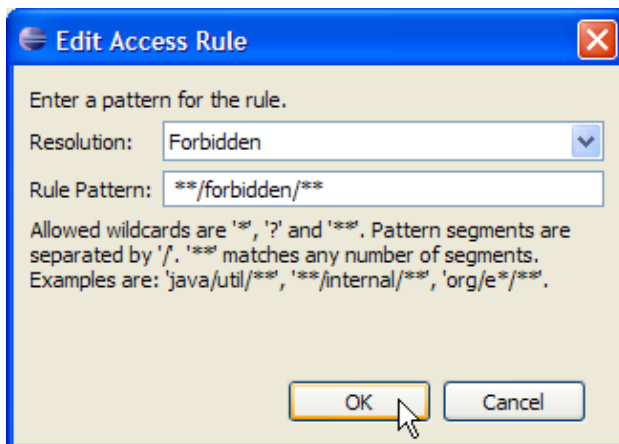
Click *Edit...*

15. In *Type Access Rules*, click *Add...*



16. In *Edit Access Rule*, select "Forbidden" for *Resolution*.

Type `**/forbidden/**` in *Rule Pattern* field.

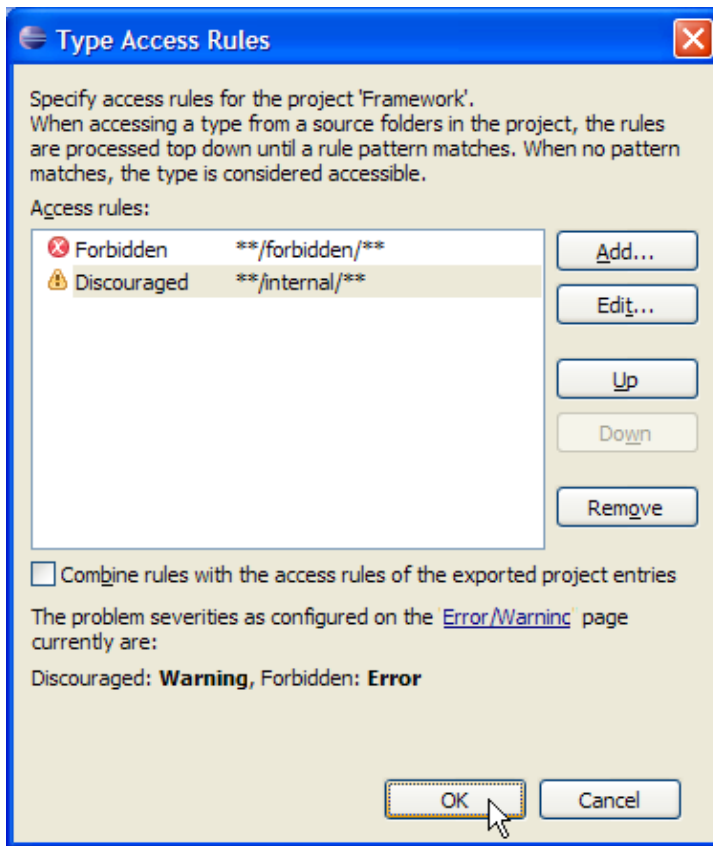


Click **OK**.

17. Add another access rule:

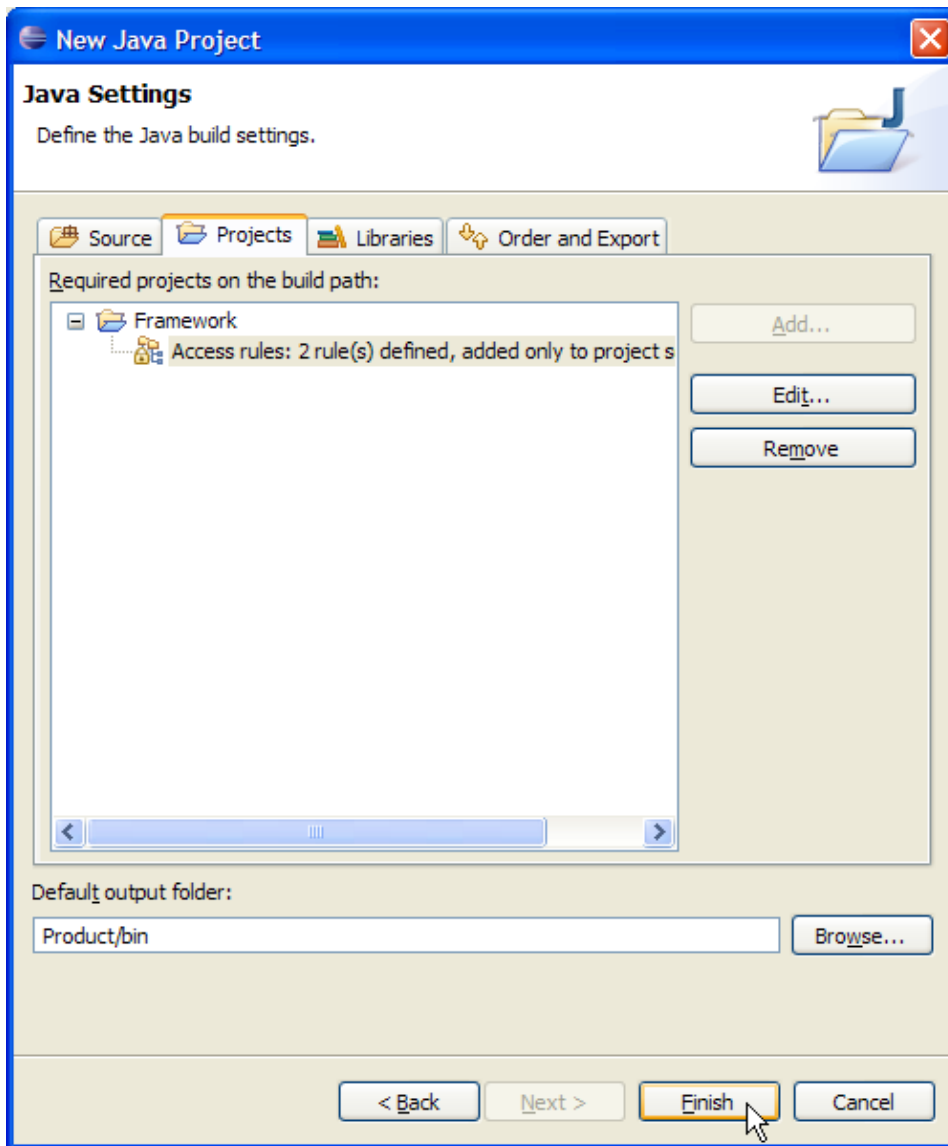
Resolution: "Discouraged" and **Rule Pattern:** `**/internal/**`.

18. Your access rules now look as follows:



Click **OK**.

19. Dependent project has now 2 access rules set.

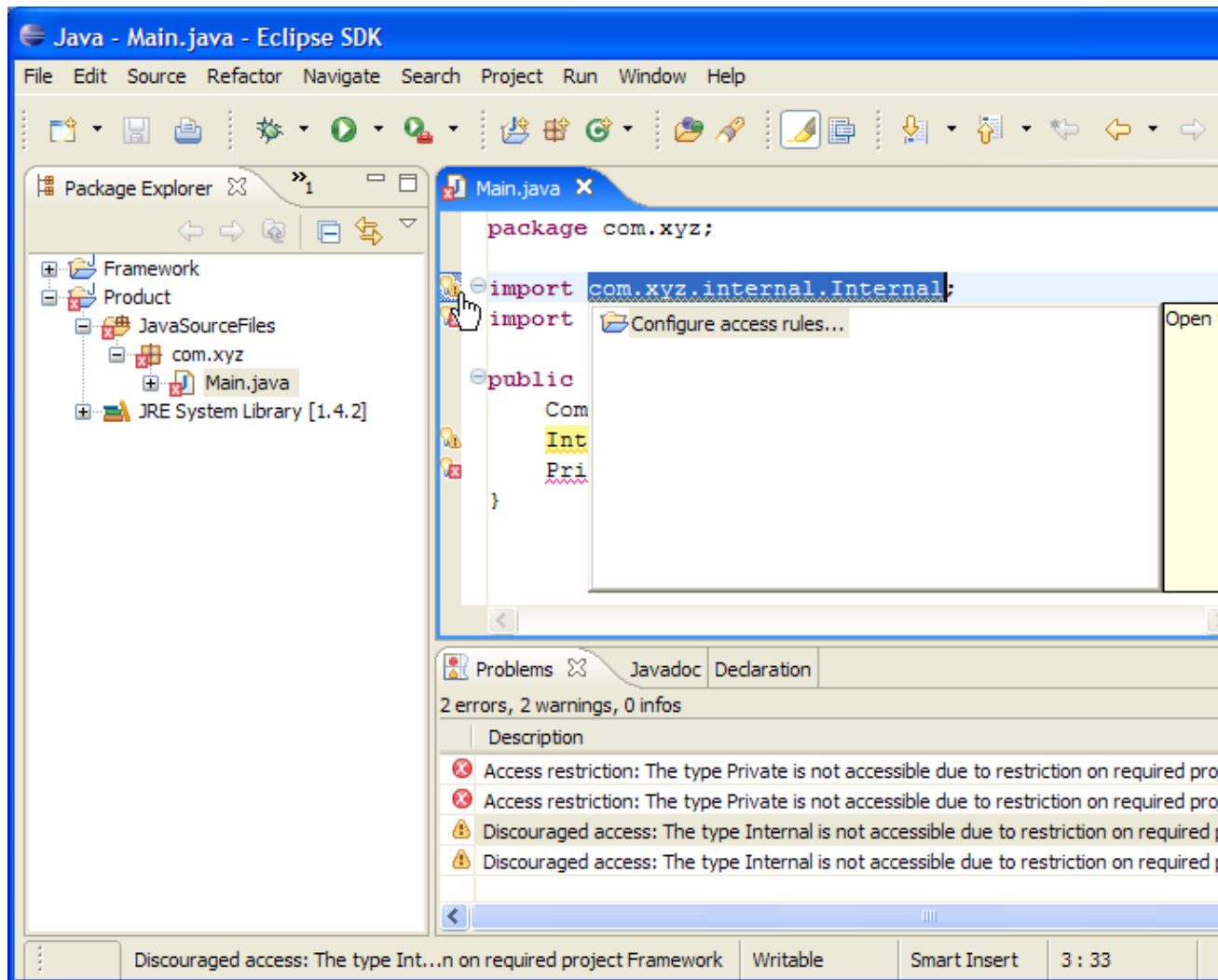


Click **Finish**.

20. You now have a Java project which contains the source of "Product" and which is using the source of "Framework".

Some packages of the project "Framework" are restricted and if you try to import them, compiler displays either warnings or errors depending on your restriction level:

Basic tutorial



Related concepts

[Java projects](#)

[Java views](#)

Related tasks

[Working with build paths](#)

[Creating a new Java project](#)

[Creating a Java project with source folders](#)

[Creating a new source folder](#)

[Using the Package Explorer](#)

Related reference

[New Java Project Wizard](#)

[Package Explorer View](#)

Getting Started with Eclipse 3.1 and J2SE 5.0

Eclipse 3.1 includes full support for the new features of J2SE 5.0 (codenamed "Tiger"). One of the most important consequences of this support is that you may not notice it at all—everything you expect to work for J2SE 1.4, including editing, compiling, debugging, quick fixes, refactorings, source actions, searching, etc., will work seamlessly with J2SE 5.0's new types and syntax. In this document, we will introduce some of the more interesting capabilities Eclipse users will find when working with J2SE 5.0.

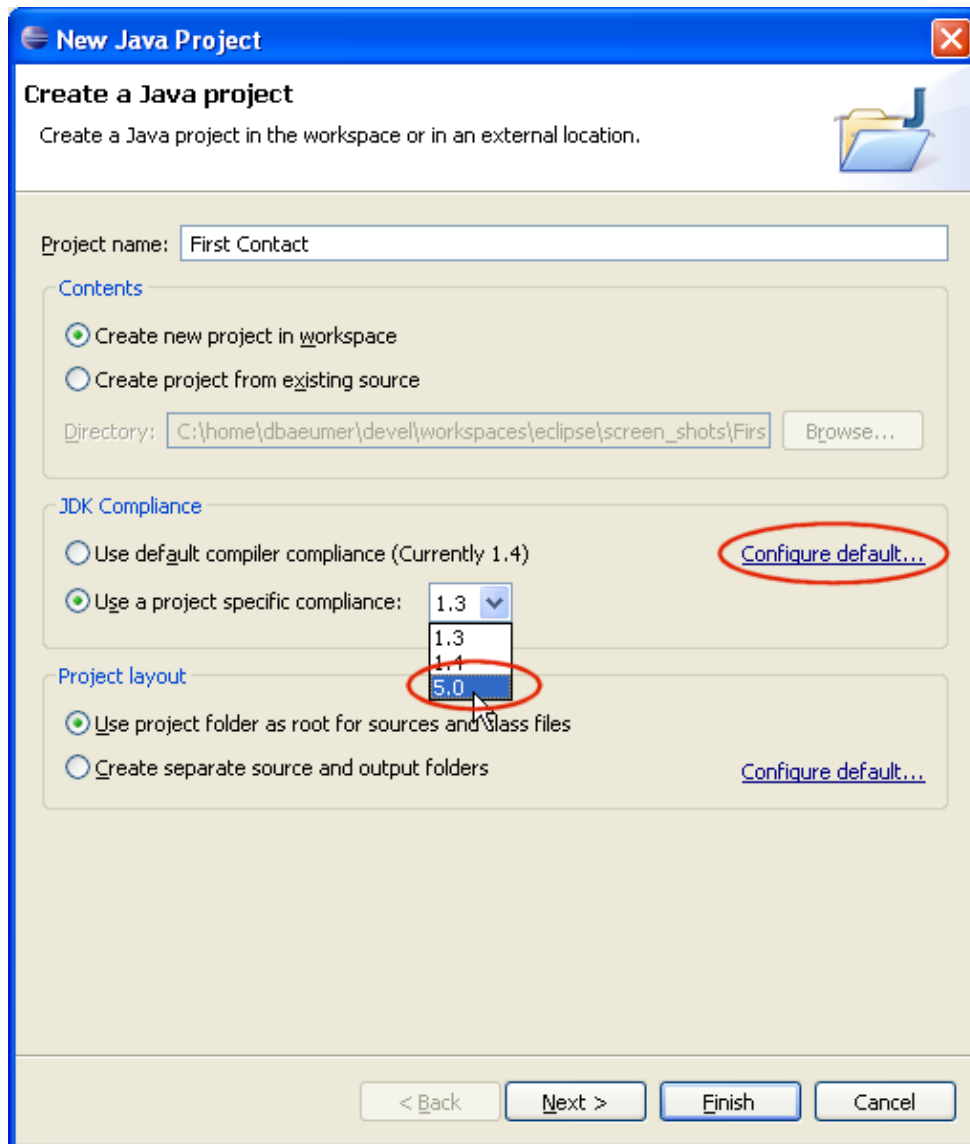
Prerequisites

In order to develop code compliant with J2SE 5.0, you will need a J2SE 5.0 Java Runtime Environment (JRE). If you start Eclipse for the first time using a J2SE 5.0 JRE, then it will use it by default. Otherwise, you will need to use the *Installed JRE's* dialog (*Windows > Preferences > Java > Installed JRE's*) to register it with Eclipse.

This document introduces some of the new language features in J2SE 5.0 very briefly, but it is not a proper tutorial for these features.

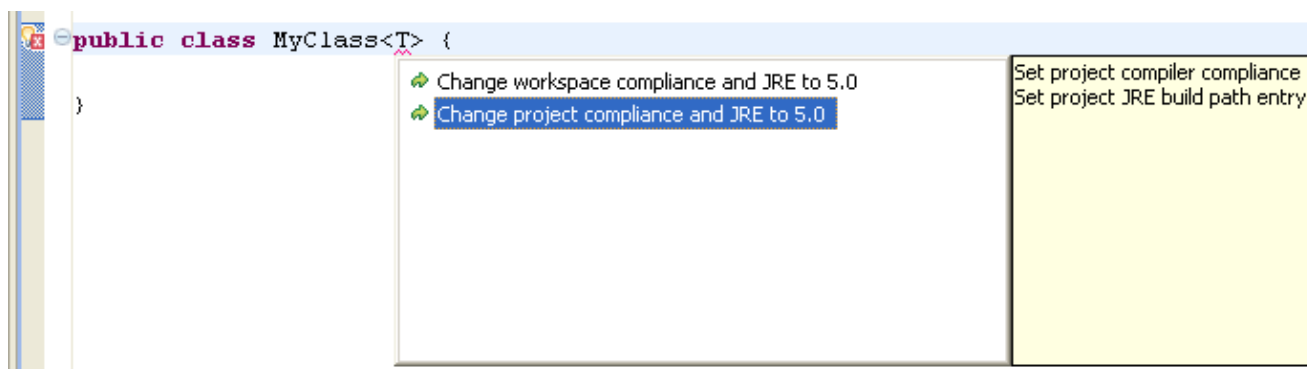
Compiler Compliance Level

To use the new J2SE 5.0 features, you must be working on a project that has a 5.0 compliance level enabled. New projects can easily be marked as 5.0-compliant on the first page of the *New > Project* wizard:



To convert an existing J2SE 1.4 project to J2SE 5.0, you can simply:

1. Make sure you have a J2SE 5.0 JRE installed.
2. Start using the 5.0 features in your code.
3. When a compiler error is flagged, use Quick Fix to update the project's compliance level:



For more fine-tuned control, the compiler compliance level can be set globally for a workspace (**Windows > Preferences > Java > Compiler**) or individually for each project (from the project's context menu, choose **Properties > Java Compiler**). Projects with different compliance levels can co-exist in the workspace, and depend on each other. You can also fine-tune the kinds of compiler warnings and errors produced for each project using **Properties > Java Compiler > Errors/Warnings > J2SE 5.0 Options**

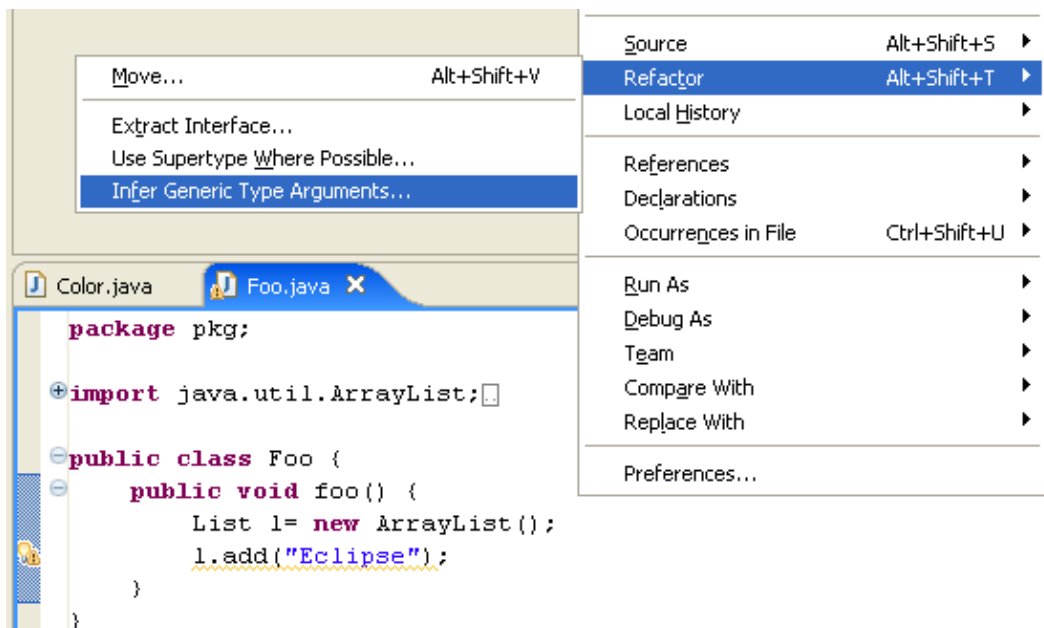
Generic Types

Generic types allow objects of the same class to safely operate on objects of different types. For example, they allow compile-time assurances that a `List<String>` always contains `Strings`, and a `List<Integer>` always contains `Integers`.

Anywhere that Eclipse handles a non-generic type, it can handle a generic type:

- Generic types can be safely renamed.
- Type variables can be safely renamed.
- Generic methods can be safely extracted from / inlined into generic code.
- Code assist can automatically insert appropriate type parameters in parameterized types.

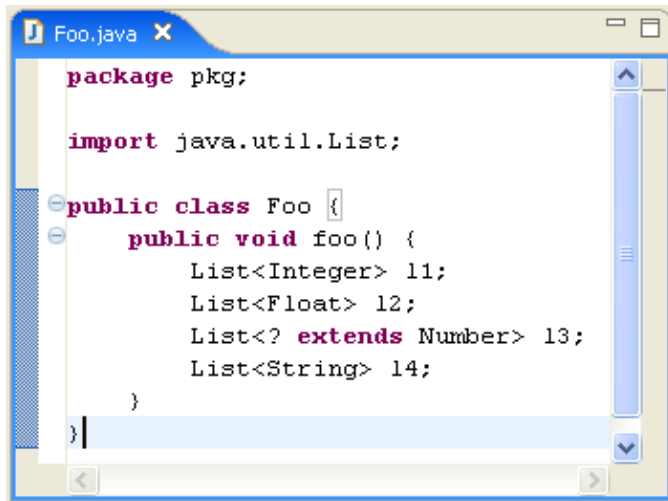
In addition, a new refactoring has been added: **Infer Generic Type Arguments** can infer type parameters for every type reference in a class, a package, or an entire project:



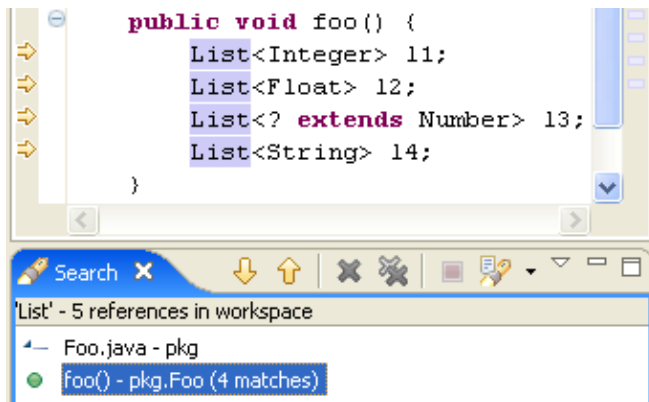
Invoking the refactoring produces:

```
public void foo() {
    List<String> l= new ArrayList<String>();
    l.add("Eclipse");
}
```

Eclipse 3.1 provides new options when searching for references to generic types. Consider this example:

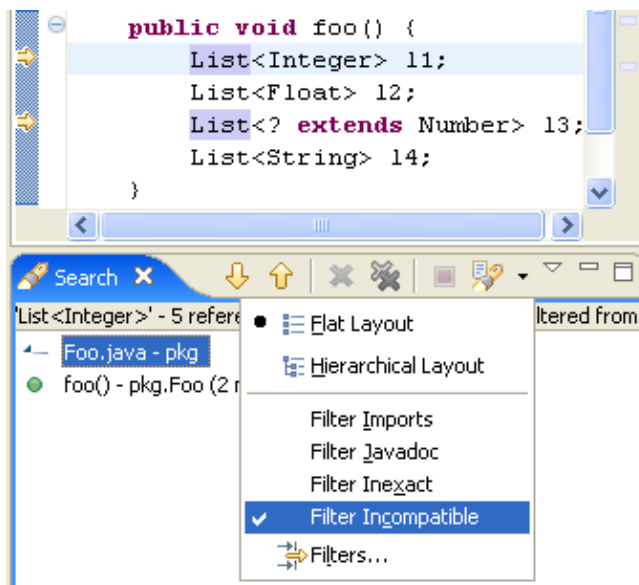


Selecting the reference to `List<Integer>` and using **Search > References > Project** will highlight the List types on all four lines:

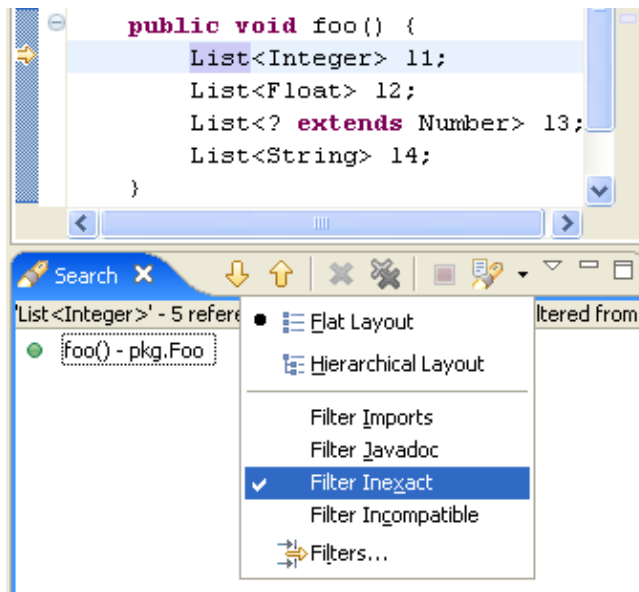


Using the Search View menu, the results can be filtered:

Filter Incompatible leaves only references to types that are assignment-compatible with the selected type:



Filter Inexact leaves only type references with the exact same signature:



Annotations

Annotations attach metadata about how Java types and methods are used and documented to the Java source and can then affect compilation or be queried at run-time. For example, `@Override` will trigger a compiler warning if the annotated method does not override a method in a superclass:

```
@Override public String toString() { return "a"; }
```

java.lang.Override

Indicates that a method declaration is intended to override a method declaration in a superclass. If a method is annotated with this annotation type but does not override a superclass method, compilers are required to generate an error message.

@author

Joshua Bloch

@since

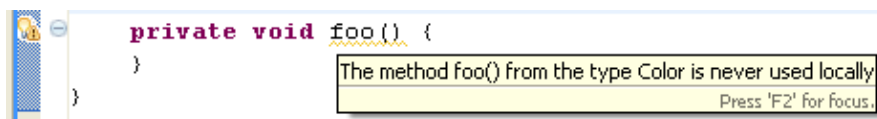
1.5

Press 'F2' for focus.

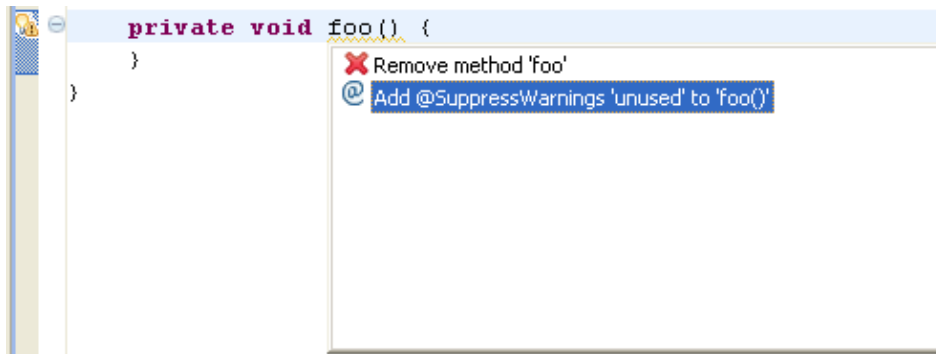
Everything you can do with a Java type, you can do with an annotation:

- Create new annotations using **New > Annotation**
- Refactor: rename, move, change signatures of members, etc.
- Search for occurrences
- Use code assist to fill in names and values

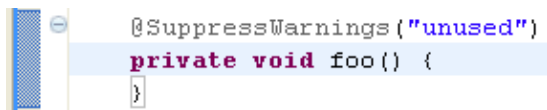
A very useful annotation with full support in Eclipse is `@SuppressWarnings`. For example, consider a private method that is currently unused, but you'd rather not delete:



Invoking quick fix on the warning proposes adding a `@SuppressWarnings` annotation:



Selecting the quick fix adds the annotation. The Eclipse compiler honors the annotation by removing the warning on `foo`:



Enumerations

Enumerations are types that are instantiated at runtime by a known, finite set of objects:

```
public enum Color {
    RED, GREEN, BLUE;
}
```

Again, anything you can do to a Java class can be done to an enumeration:

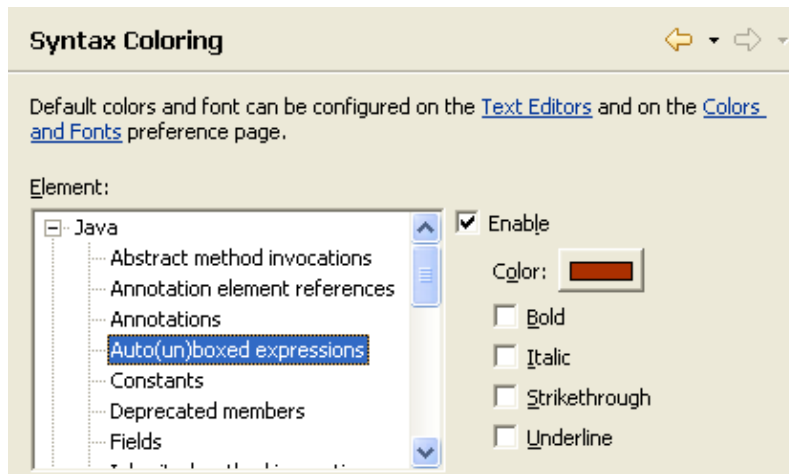
- Create new enumerations using *New > Enum*
- Refactor: rename, move, rename constants, etc.
- Search for occurrences
- Use code assist to fill in constants

Autoboxing

Autoboxing and auto unboxing allow for elegant syntax when primitive types are assigned to or retrieved from Object references:

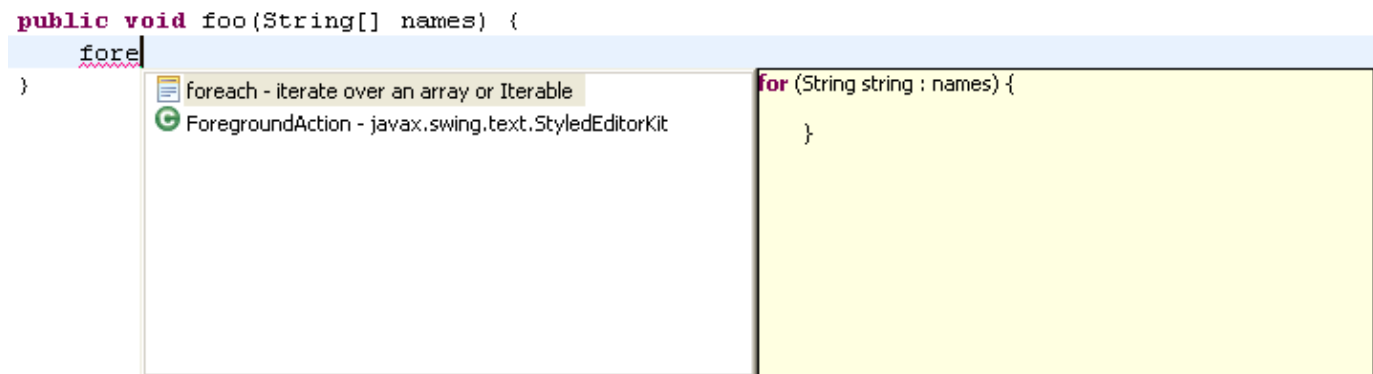
```
public void foo(Integer i) {
    foo(10);
}
```

Eclipse's source manipulation features handle autoboxing seamlessly, giving the correct types to new local variables and correct code assists. For code understanding, it is also possible to flag instances of autoboxing or autounboxing as compile warnings (*Window > Preferences > Java > Compiler > Errors/Warnings > J2SE 5.0 Options > Boxing and unboxing conversions*), or highlight them using syntax coloring (*Window > Preferences > Java > Editor > Syntax Coloring > Java > Auto(un)boxed expressions*):



Enhanced for loop

For the common case of operating on each element of an array or collection in turn, J2SE 5.0 allows a new, cleaner syntax. Eclipse 3.1 provides a "foreach" code template that can automatically guess the collection to be iterated:



Choosing the template produces:

```
public void foo(String[] names) {
    for (String string : names) {
    }
}
```

Eclipse 3.1 also provides a "Convert to enhanced for loop" quick-assist to upgrade 1.4-style `for` loops where possible.

Other

All other features of J2SE 5.0 are handled flexibly by Eclipse's editing, searching, and code manipulation tools:

- Static imports

Basic tutorial

- Varargs
- Covariant return types

Happy coding!

Creating a new Java Scrapbook Page

You can create a new Java scrapbook page using any one of several different approaches. [See Creating a Scrapbook Page](#)

Parameters page

Extract Method

Method name:

Access modifier: ☐ public ☐ protected ☐ default ☒ private

Parameters:

Type	Name
TestResult	result

☐ Add thrown runtime exceptions to method signature
☐ Generate method comment
☐ Replace duplicate code fragments

Method signature preview:
`private void runTest(TestResult result)`

Extract Method Parameters Page

- In the **Method name** field, type a name for the new method that will be extracted.
- In the **Access Modifier** list, specify the method's visibility (public, default, protected, or private).
- You can **Add thrown runtime exceptions to method signature** by selecting the corresponding checkbox.
- You can **Generate method comment** by selecting the corresponding checkbox.
- You can rearrange and rename the parameters for the new method.
- Click **OK** to perform a quick refactoring, or click **Preview** to perform a controlled refactoring.

Related Topics:

- [See Refactoring without Preview](#)
- [See Refactoring with Preview](#)

Problems page

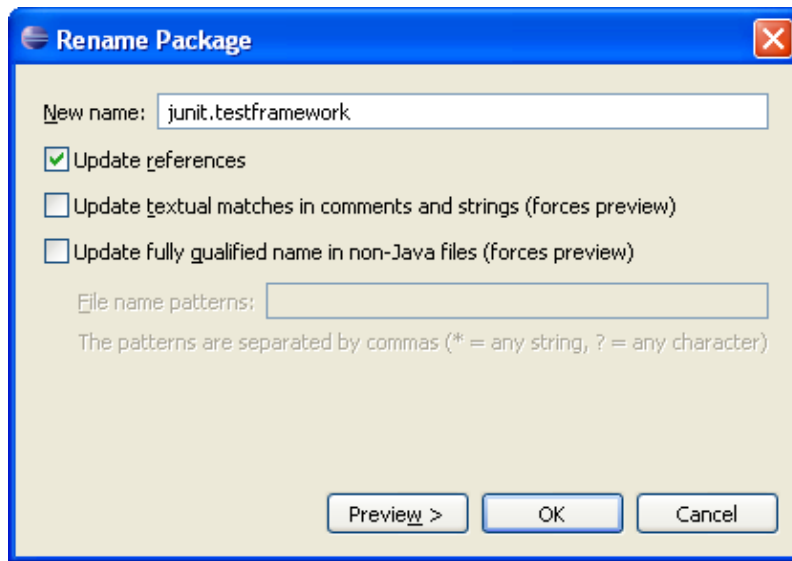
The most common mistakes when extracting a method are:

- The selection does not cover statements or an expression from a method body.
- The selection does not cover a whole set of statements or an expression.

You can use the *Edit > Expand Selection To* actions to expand an selection to a valid expression or set of statements.

On the problems pages, you can press F1 to link to a detailed description of errors.

Parameters page



Rename Package Parameters Page

- In the **Enter new name** field, type a new name for the package.
- If you do not want to update references to the renamed package, deselect the **Update references** checkbox.
- If you want to update references to the renamed package and strings in code, select the **Update textual matches in comments and strings** checkbox.
- If you want to update references in non-Java files (like XML configuration files), select the **Update fully-qualified name in non-Java files** checkbox.
- Click **OK** to perform a quick refactoring, or click **Preview** to perform a controlled refactoring.

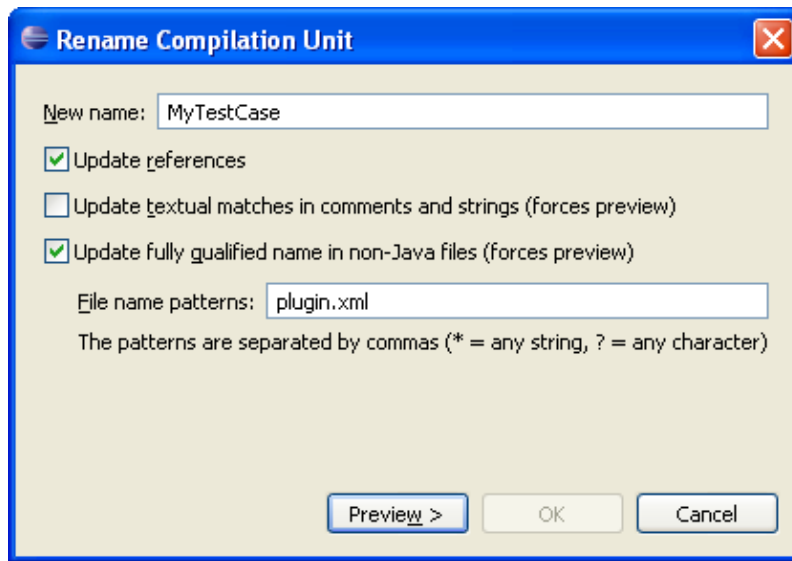
Note: References in Javadoc comments, regular comments and string literals are updated based on textual matching. It is recommended that you perform a controlled refactoring and review the suggested changes if you select one of these options.

■ Related tasks

[See Refactoring without Preview](#)

[See Refactoring with Preview](#)

Parameters page



Rename Compilation Unit Parameters Page

- In the **Enter new name** field, type a new name for the compilation unit.
- If you do not want to update references to the renamed compilation unit, deselect the **Update references** checkbox.
- If you want to update references to the renamed compilation unit and strings in code, select the **Update textual matches in comments and strings** checkbox.
- If you want to update references in non-Java files (like XML configuration files), select the **Update fully-qualified name in non-Java files** checkbox.
- Click **OK** to perform a quick refactoring, or click **Preview** to perform a controlled refactoring.

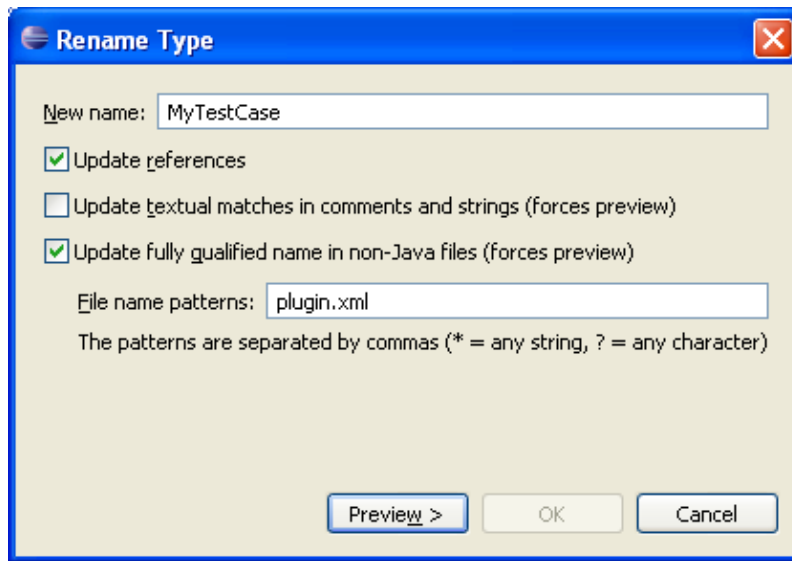
Note: References in Javadoc comments, regular comments and string literals are updated based on textual matching. It is recommended that you perform a controlled refactoring and review the suggested changes if you select one of these options.

■ Related tasks

[See Refactoring without Preview](#)

[See Refactoring with Preview](#)

Parameters page



Rename Type Wizard Page

- In the **Enter new name** field, type a new name for the type.
- If you do not want to update references to the renamed type, deselect the **Update references to the renamed element** checkbox.
- If you want to update textual references in strings and comments which are referring to the renamed type, select the **Update textual matches in comments and strings** checkbox.
- If you want to update fully qualified names in non-Java files, select the **Update fully qualified name in non-Java files** checkbox.
- Click **OK** to perform a quick refactoring, or click **Preview** to perform a controlled refactoring.

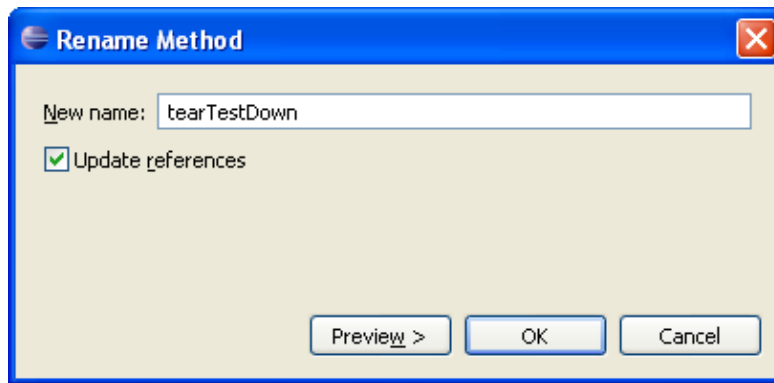
Note: References in comments and strings are updated based on textual matching. It is recommended that you perform a controlled refactoring and review the suggested changes if you select one of these options.

■ Related tasks

[Refactoring without Preview](#)

[Refactoring with Preview](#)

Parameters page



Parameters Page for the Rename Method Refactoring Command

- In the *Enter new name* field, type a new name for the method.
- If you do not want to update references to the renamed method, deselect the *Update references to the renamed element* checkbox.
- Click **OK** to perform a quick refactoring, or click **Preview** to perform a controlled refactoring.

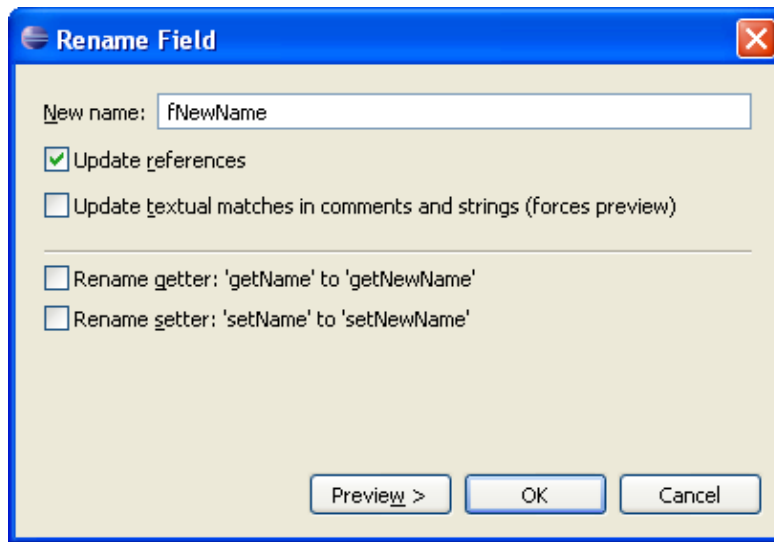
■ Related tasks

[See Refactoring without Preview](#)

[See Refactoring with Preview](#)

[See Showing a Type's Compilation Unit in the Packages View](#)

Parameters page



Parameters Page for the Rename Field Refactoring Command

- In the **Enter new name** text field, type a new name for the field that you're renaming.
- If you do not want to update references to the renamed field, deselect the **Update references to the renamed element** checkbox.
- If you want to update textual references in strings and comments which are referring to the renamed field, select the **Update textual matches in comments and strings** checkbox.
- If the refactoring finds accessor (getter/setter) methods to the field you're renaming, it offers you to rename them as well (and update all references to them):
 - ◆ If you want to rename the getter, select the **Rename Getter** checkbox
 - ◆ If you want to rename the setter, select the **Rename Setter** checkbox
- Click **OK** to perform a quick refactoring, or click **Preview** to perform a controlled refactoring.

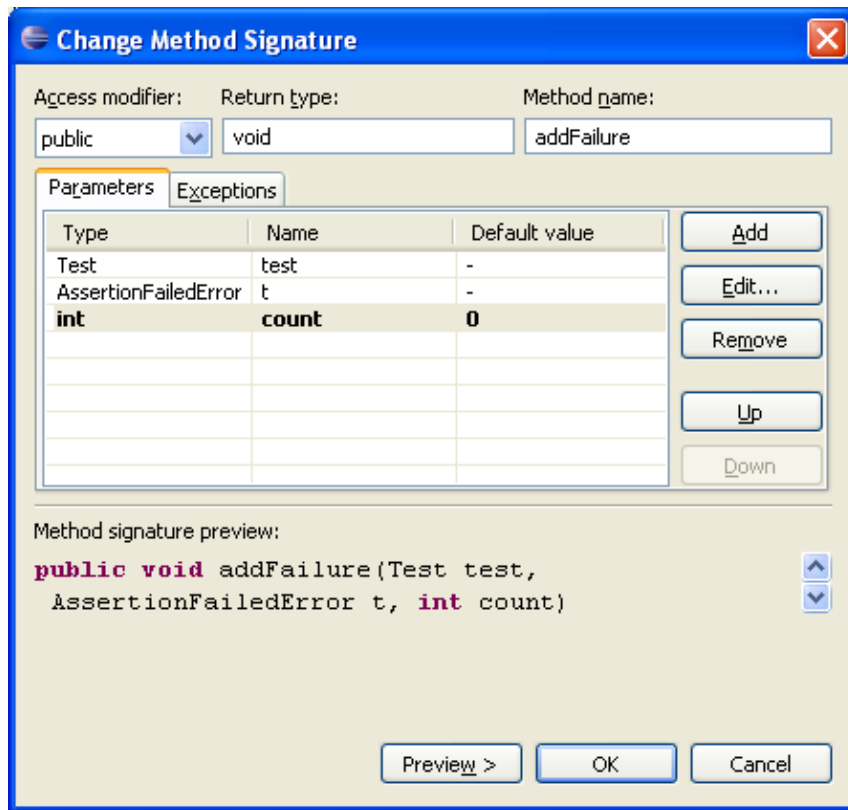
Note: The refactoring detects getters / setters using preferences set on **Window > Preferences > Java > Code Style** preference page.

Note: References in Javadoc comments, regular comments and string literals are updated based on textual matching. It is recommended that you perform a controlled refactoring and review the suggested changes if you select one of these options.

Related Topics:

- [See Refactoring without Preview](#)
- [See Refactoring with Preview](#)

Parameters page



The dialog box is titled "Change Method Signature". It has three input fields at the top: "Access modifier:" with a dropdown menu showing "public", "Return type:" with a text field showing "void", and "Method name:" with a text field showing "addFailure". Below these are two tabs: "Parameters" (selected) and "Exceptions". The "Parameters" tab contains a table with three columns: "Type", "Name", and "Default value". The table has three rows: the first row has "Test", "test", and "-"; the second row has "AssertionFailedError", "t", and "-"; the third row has "int", "count", and "0". To the right of the table are five buttons: "Add", "Edit...", "Remove", "Up", and "Down". Below the table is a "Method signature preview:" section with a text area showing the signature: `public void addFailure(Test test, AssertionFailedError t, int count)`. At the bottom are three buttons: "Preview >", "OK", and "Cancel".

Type	Name	Default value
Test	test	-
AssertionFailedError	t	-
int	count	0

Method signature preview:

```
public void addFailure(Test test,
    AssertionFailedError t, int count)
```

Parameters Page for the Change Method Signature Refactoring Command

- Click in the *Name* column on the row containing the parameter you want to change or select the row and press *Edit* and type a new name for the parameter.

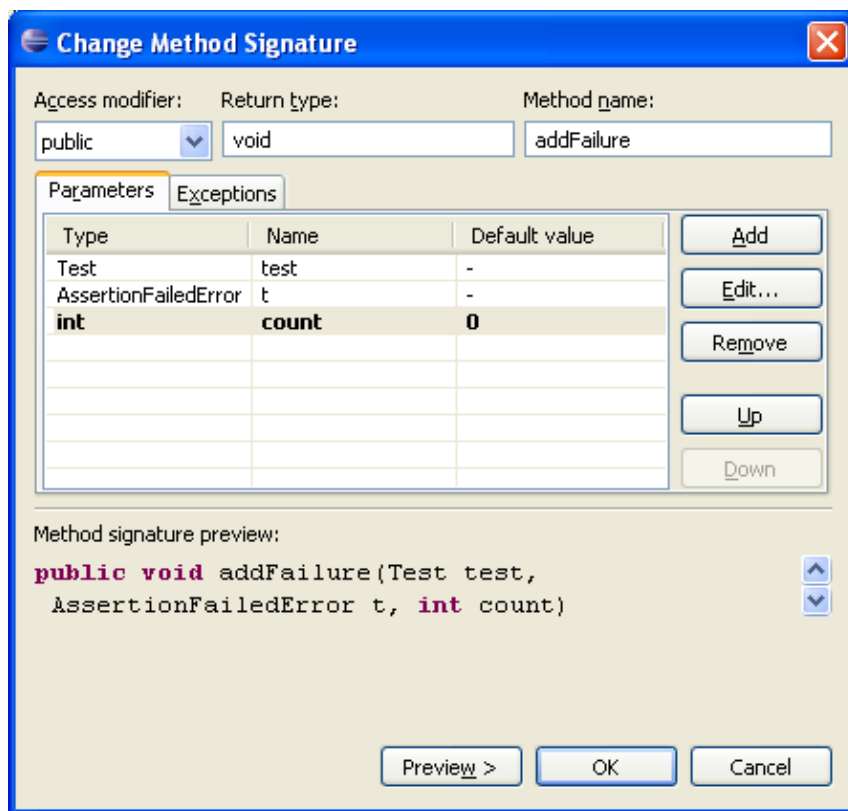
Related tasks

[See Refactoring without Preview](#)

[See Refactoring with Preview](#)

[See Showing a Type's Compilation Unit in the Packages View](#)

Parameters page



Parameters Page for the Change Method Signature Refactoring Command

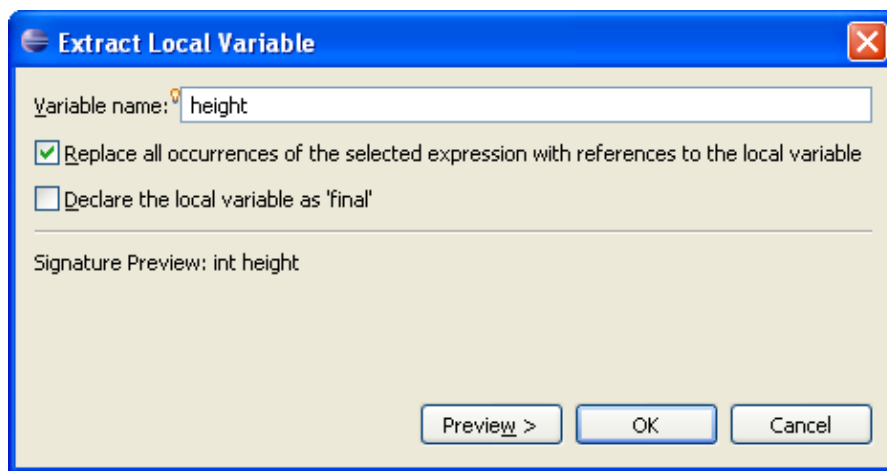
- Use the Access Modifier drop-down to control the method's visibility
- Change the method's return type or name by editing the provided text fields
- Select one or more parameters and use the **Up** and **Down** buttons to reorder the parameters (you can see a signature preview below the parameter list)
- Use the **Add** button to add a parameter; you can then edit its type, name and default value in the table
- Switch to the **Exceptions** tab to add or remove thrown exceptions
- Press **Preview** to see the preview or **OK** to perform the refactoring without seeing the preview

This refactoring changes the signature of the selected method and all methods that override it. Also, all references will be updated to use the signature.

Related Topics:

- [See Refactoring without Preview](#)
- [See Refactoring with Preview](#)

Parameters page



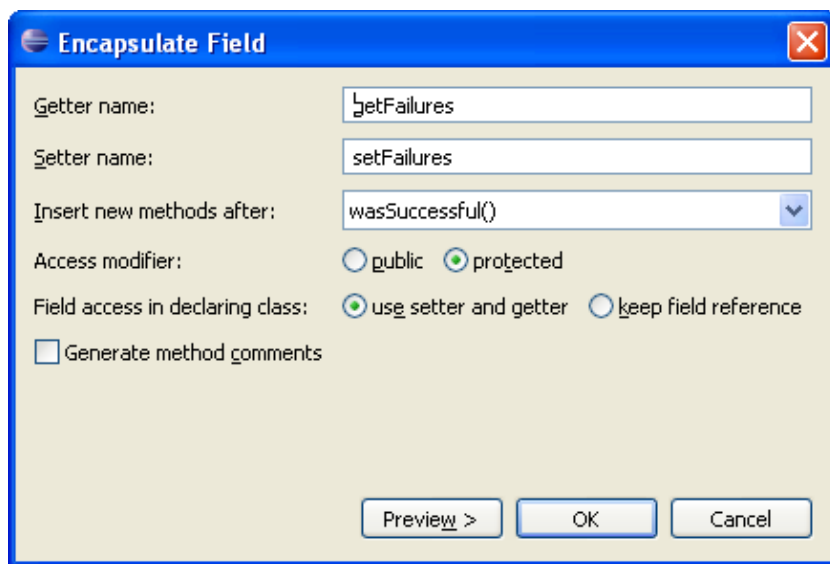
Parameters Page for the Extracting Local Variable Refactoring Command

- In the Variable name field, enter a name for the extracted variable
- Optionally, clear the ***Replace all occurrences of the selected expression with references to the local variable*** checkbox if you want to replace only the expression you selected when invoking the refactoring.
- Optionally, select ***Define the local variable as 'final'***
- Press ***Preview*** to see the preview of the changes or ***OK*** to perform the refactoring without preview

Related Topics:

- [See Refactoring without Preview](#)
- [See Refactoring with Preview](#)

Parameters page



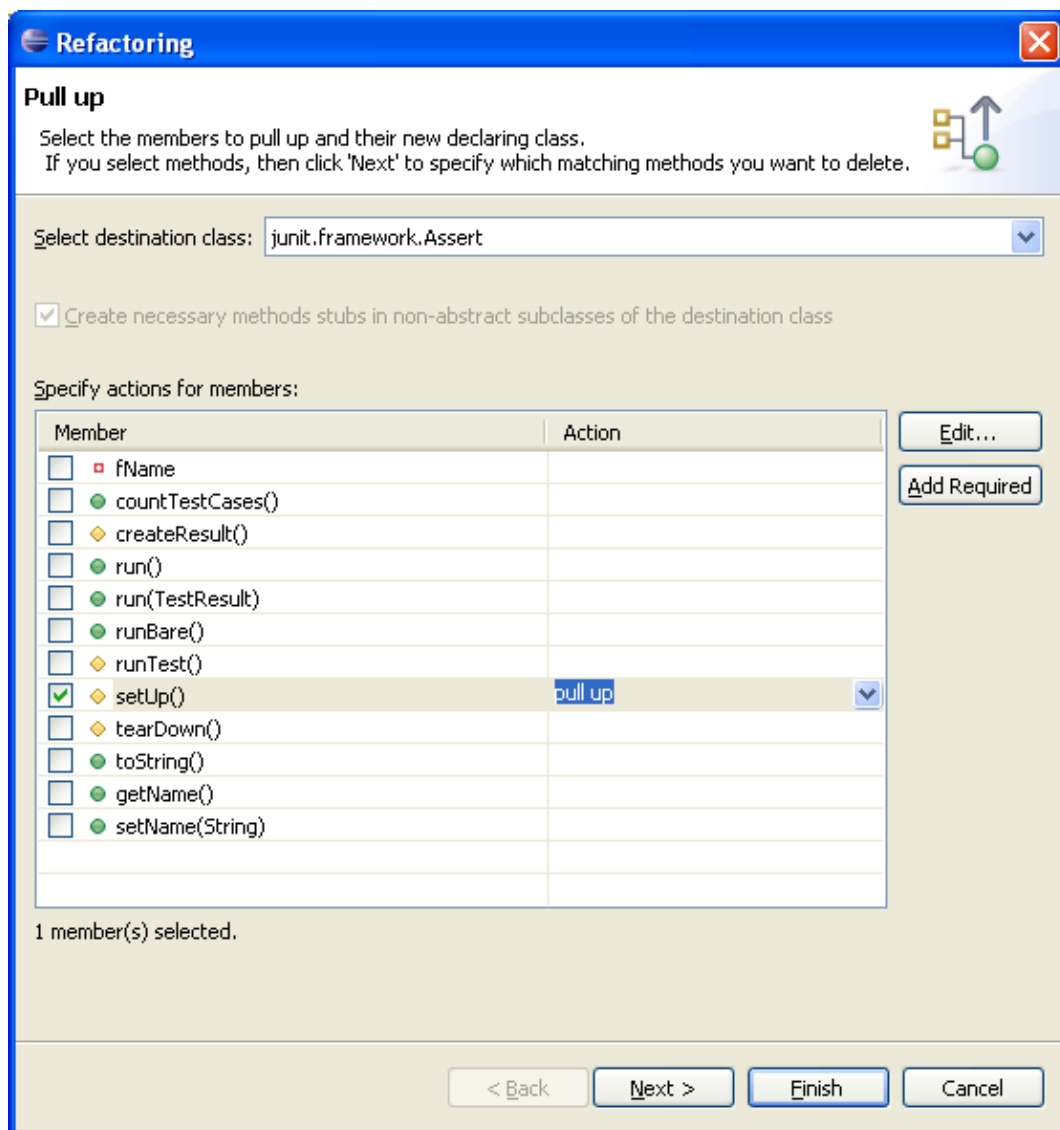
Parameters Page for the Self Encapsulate Field Refactoring Command

- In the **Getter name** field, enter the name for the getter.
- In the **Setter name** field, enter the name for the setter.
- Use the **Insert new method after** combo-box to indicate the location for the getter and/or setter methods.
- Select one radio button from the **Access modifier** group to specify the new method's visibility.
- In the class in which the field is declared, read and write accesses can be direct or you can use getter and setter.
 - ◆ Select the **use getter and setter** radio button if you want the refactoring to convert all these accesses to use getter and setter.
 - ◆ Select the **keep field reference** radio button if you do not want the refactoring to modify the current field accesses in the class in which the field is declared.
- Press **Preview** to perform refactoring with preview or press **OK** to perform refactoring without preview.

Related Topics:

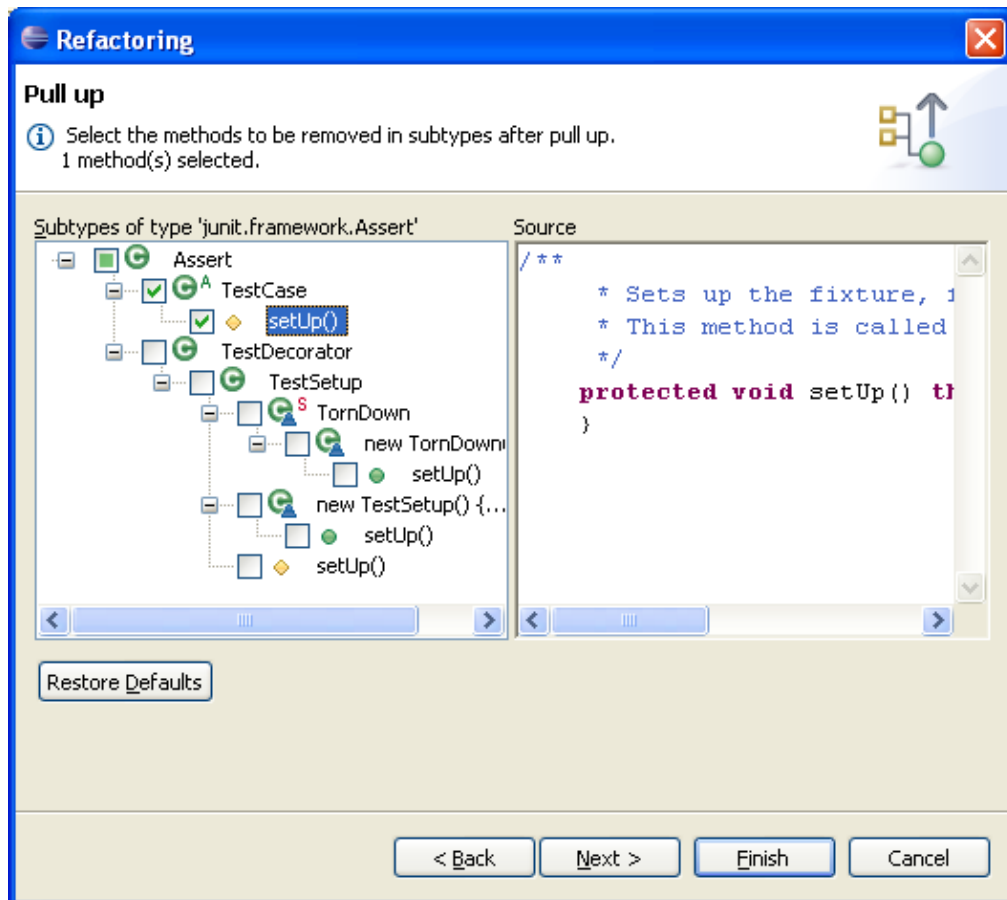
- [See Refactoring without Preview](#)
- [See Refactoring with Preview](#)

Parameters page



Parameters Page for the Pulling members up to superclass Refactoring Command

- Select the destination class
- In the list, select the members that you want to pull up or declare abstract
- Press the Edit button to specify the action that you want to perform for the selected members (you can also edit the table cells in-place.)
- Press **Next** to see the next page or press **Finish** to perform the refactoring

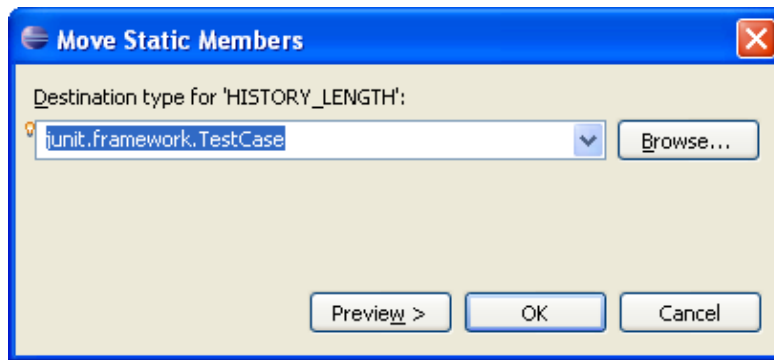


- In the left pane, select the methods that you want to be deleted after pull up (so that the superclass implementation can be used instead).
Note: the methods originally selected when invoking the refactoring are pre-selected in the left pane
- Press **Next** to see the preview or press **Finish** to perform the refactoring

Related Topics:

- [See Refactoring without Preview](#)
- [See Refactoring with Preview](#)

Parameters page



Parameters Page for the Moving static members between types Refactoring Command

- Use the text field to enter the destination type name or press the ***Browse*** button to see a list of types.
- Press ***Preview*** to see a preview or press ***OK*** to perform the refactoring without preview.

Related Topics:

- [See Refactoring without Preview](#)
- [See Refactoring with Preview](#)

Building circular projects

To enable building circular projects:

Select the *Window > Preferences > Java > Compiler > Building* page.
Then set the option *Circular dependencies* to *Warning*.

To disable building circular projects:

Select the *Window > Preferences > Java > Compiler > Building* page.
Then set the option *Circular dependencies* to *Error*.

To enable building a single project involved in a cycle:

Select the project, and from its pop-up menu, select *Properties*.
In the Properties dialog, select the *Java Compiler > Building* page.
Then set the option *Circular dependencies* to *Warning*.

To disable building a single project involved in a cycle:

Select the project, and from its pop-up menu, select *Properties*.
In the Properties dialog, select the *Java Compiler > Building* page.
Then set the option *Circular dependencies* to *Error*.

■ Related concepts

[Java builder](#)

[Build class path](#)

■ Related tasks

[Building a Java program](#)

[Building manually](#)

[Viewing compilation errors and warnings](#)

[Working with build paths](#)

[Adding a JAR file to the build path](#)

[Adding a library folder to the build path](#)

[Viewing and editing a project's build path](#)

■ Related reference

[Java Build path](#)

Building without cleaning the output location

To build projects without cleaning the output location:

Select the *Window > Preferences > Java > Compiler > Building* page.
Then set the *Scrub output folders on full build* checkbox.

To build projects after cleaning the output location:

Select the *Window > Preferences > Java > Compiler > Building* page.
Then clear the *Scrub output folders on full build* checkbox.

To build a single project without cleaning the output location:

Select the project, and from its pop-up menu, select *Properties*.
In the Properties dialog, select the *Java Compiler > Building* page.
Then set the *Scrub output folders on full build* checkbox.

To build a single project after cleaning the output location:

Select the project, and from its pop-up menu, select *Properties*.
In the Properties dialog, select the *Java Compiler > Building* page.
Then clear the *Scrub output folders on full build* checkbox.

■ Related concepts

[Java builder](#)

[Build class path](#)

■ Related tasks

[Building a Java program](#)

[Building manually](#)

[Viewing compilation errors and warnings](#)

[Working with build paths](#)

[Adding a JAR file to the build path](#)

[Adding a library folder to the build path](#)

[Viewing and editing a project's build path](#)

■ Related reference

[Java Build path](#)

Attaching source to a library folder

You can attach source to a library folder to enable source-level stepping and browsing of classes contained in a library folder. Unless its source code is attached to a library folder in the workbench, you will not be able to view the source for the library folder.

To attach source to a library folder:

1. Select the project, and from its pop-up menu, select **Properties**.
In the Properties dialog, select the Java Build Path page.
2. On the **Libraries** tab, select the library folder to which you want to attach source.
Expand the node by clicking on the plus and select the node *Source Attachment*. Click the **Edit** button to bring up the source attachment dialog.
3. Fill in the **Location path** field depending on the location, choose between the workspace, an external file or external folder.
4. Click **OK**.

or

1. Select the library folder in the Package Explorer, and from its pop-up menu, select **Properties**.
In the Properties dialog, select the **Java Source Attachment** page.
2. Fill in the **Location path** field depending on the location, choose between the workspace, an external file or external folder.
3. Click **OK**.

■ Related concepts

[Java development tools \(JDT\)](#)

■ Related tasks

[Attaching source to variables](#)

[Creating a new JAR file](#)

[Stepping through the execution of a program](#)

■ Related reference

[Java Build Path](#)

[Source Attachment dialog](#)

Launching a Java applet

If your Java program is structured as an applet, you can use the *Java Applet* launch configuration. This launch configuration uses information derived from the workbench preferences and your program's Java project to launch the program.

1. In the Package Explorer, select the Java compilation unit or class file you want to launch.
2. From the pop-up menu, select **Run > Java Applet**. Alternatively, select **Run > Run As > Java Applet** in the workbench menu bar, or select **Run As > Java Applet** in the drop-down menu on the **Run** tool bar button.
3. Your program is now launched.

You can also launch a Java applet by selecting a project instead of the compilation unit or class file. You will be prompted to select a class from those classes that extend Applet. (If only one applet class is found in the project, that class is launched as if you selected it.)

■ Related concepts

[Debugger](#)

■ Related tasks

[Re-launching a program](#)

[Running and debugging](#)

[Stepping through the execution of a program](#)

■ Related reference

[Debug view](#)

[Package Explorer](#)

Launching a Java program in debug mode

Launching a program in debug mode allows you to suspend and resume the program, inspect variables, and evaluate expressions using the debugger.

To launch a Java program in debug mode,

1. In the Package Explorer, select the Java compilation unit or class file you want to launch.
2. Select **Run > Debug As > Java Application**.
or Select **Debug As > Java Application** in the drop-down menu on the **Debug** tool bar button.
3. Your program is now launched and the launched process appears in the Debug view.

If you want your program to stop in the *main* method so that you can step through its complete execution, create a **Java Application** launch configuration and check the **Stop in main** checkbox on the **Main** tab.

You can also debug a Java program by selecting a project instead of the compilation unit or class file. You will be prompted to select a class from those classes that define a *main* method. (If only one class with a main method is found in the project, that class is launched as if you selected it.)

■ Related concepts

[Java views](#)

[Java editor](#)

[Debugger](#)

■ Related tasks

[Connecting to a remote VM with the Java Remote Application launcher](#)

[Re-launching a program](#)

[Running and debugging](#)

[Setting execution arguments](#)

[Stepping through the execution of a program](#)

■ Related reference

[Debug view](#)

[Package Explorer](#)

Inspecting values

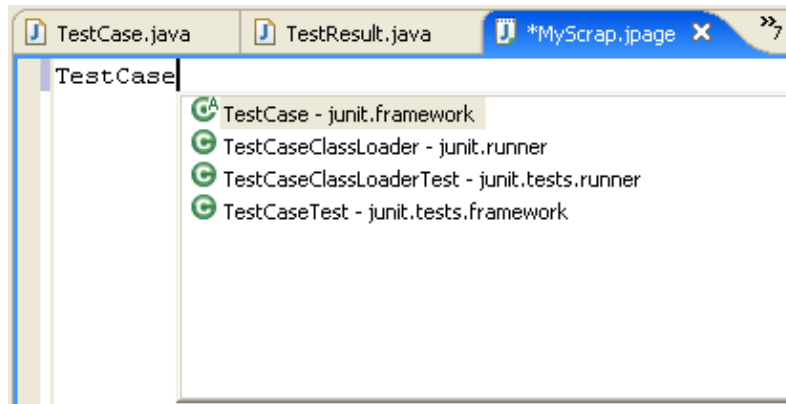
When stack frame is selected, you can see the visible variables in that stack frame in the Variables view.

The Variables view shows the value of primitive types. Complex variables can be examined by expanding them to show their members.

Using code assist

The scrapbook editor supports code assist similarly to the regular Java editor.

For example, type *TestCase* in the scrapbook editor and press **Ctrl+Space**. Code assist prompts you with possible completions.



■ Related reference

[Java Content Assist](#)

Scrapbook error reporting

Java scrapbook errors are reported in the scrapbook page editor.

■ Related tasks

[Viewing compilation errors](#)

[Viewing runtime exceptions](#)

Viewing compilation errors

If you try to evaluate an expression containing a compilation error, it will be reported in the scrapbook editor.

For example, type and select the (invalid) expression `System.println("hi")` in the editor and click ***Execute*** in the toolbar.

The error message *The method println(java.lang.String) is undefined for the type java.lang.System* appears in the editor at the point of the error.

Go to file for breakpoint

If the resource containing the selected breakpoint is not open and/or active, this command causes the file to be opened and made active, focusing on the line with which the breakpoint is associated.

■ Related concepts

[Breakpoints](#)

■ Related tasks

[Adding breakpoints](#)

[Removing breakpoints](#)

[Launching a Java program](#)

[Running and debugging](#)

Add Java exception breakpoint

This command allows you to add a Java exception breakpoint. In the resulting dialog:

- In the *Choose an Exception* field, type a string that is contained in the name of the exception you want to add. You can use wildcards as needed ("*" for any string and "?" for any character).
- In the exceptions list, select the exception you want to add.
- Check or clear the *Caught* and *Uncaught* checkboxes as needed to indicate on which exception type you want to suspend the program.

Related concepts

[Breakpoints](#)

Related tasks

[Catching Java exceptions](#)

[Adding breakpoints](#)

[Removing breakpoints](#)

[Launching a Java program](#)

[Running and debugging](#)

Suspend policy

This action toggles the suspend policy of a breakpoint between suspending all of the threads in the VM and the thread in which the breakpoint occurred.

■ Related concepts

[Breakpoints](#)

■ Related tasks

[Adding breakpoints](#)

[Removing breakpoints](#)

[Launching a Java program](#)

[Running and debugging](#)

Hit count

This option sets the hit count for the selected breakpoint. The hit count keeps track of the number of times that the breakpoint is hit. When the breakpoint is hit for the n th time, the thread that hit the breakpoint suspends. The breakpoint is disabled until either it is re-enabled or its hit count is changed.

■ Related concepts

[Breakpoints](#)

■ Related tasks

[Adding breakpoints](#)

[Removing breakpoints](#)

[Launching a Java program](#)

[Running and debugging](#)

Uncaught

When this option is turned on, execution stops when the exception is thrown and is not caught in the program.

■ Related concepts

[Breakpoints](#)

■ Related tasks

[Catching exceptions](#)

[Adding breakpoints](#)

[Removing breakpoints](#)

[Launching a Java program](#)

[Running and debugging](#)

■ Related reference

[Add Java exception breakpoint](#)

Caught

When this option is turned on, execution stops when the exception is thrown and is caught in the program.

■ Related concepts

[Breakpoints](#)

■ Related tasks

[Catching exceptions](#)

[Adding breakpoints](#)

[Removing breakpoints](#)

[Launching a Java program](#)

[Running and debugging](#)

■ Related reference

[Add Java exception breakpoint](#)

Modification

When this option is turned on, the watchpoint causes execution to suspend on modification of a field.

■ Related concepts

[Breakpoints](#)

■ Related tasks

[Adding breakpoints](#)

[Removing breakpoints](#)

[Launching a Java program](#)

[Running and debugging](#)

Access

When this option is turned on, the watchpoint causes execution to suspend on access of a field.

■ Related concepts

[Breakpoints](#)

■ Related tasks

[Adding breakpoints](#)

[Removing breakpoints](#)

[Launching a Java program](#)

[Running and debugging](#)

Exit

When this option is turned on, the breakpoint causes execution to suspend on exit of the method.

■ Related tasks

[Setting method breakpoints](#)

Entry

When this option is turned on, the breakpoint causes execution to suspend on entry of the method.

Related tasks

[Setting method breakpoints](#)

Select all

This command selects all breakpoints in the *Breakpoints view*

Enable

This command enables the selected breakpoints.

Disable

This command disables the selected breakpoints. A disabled breakpoint does not cause the execution of a program to be suspended.

Remove selected breakpoint

This command removes the selected breakpoint(s).

Remove all breakpoints

This command removes all breakpoints in the workbench.

Show qualified names

This option can be toggled to show or hide qualified names.

Show supported breakpoints

When this action is toggled on, the breakpoints view only displays breakpoints applicable to the selected debug target.

Properties

This action realizes a breakpoint properties dialog for the currently selected breakpoint

Related tasks

[Adding breakpoints](#)

[Applying hit counts](#)

[Catching Java exceptions](#)

[Removing breakpoints](#)

[Enabling and disabling breakpoints](#)

[Managing conditional breakpoints](#)

[Setting method breakpoints](#)

Copy

This command copies all selected text from the Console view onto the clipboard.

Select all

This command selects all text in the current pane of the Console view.

Find/Replace

This command allows you to search for an expression and replace it with another expression.

Go to line

This command allows you to go to the specified line in the console. The line is specified in the resulting dialog.

Clear

This command clears all content in the Console view.

Terminate

This command terminates the process that is currently associated with the console

Inspect

You can type an expression in the Display view and then use the ***Inspect*** command to evaluate the expression and inspect the result in the Expressions view.

Display

You can type an expression in the Display view and then use the ***Display*** command to display its value.

Clear the display

This command clears the display view.

Select all

This command selects all expressions in the *Expressions view*

Copy variables

This command copies a text representation of all selected expressions and variables onto the clipboard.

Remove selected expressions

This command removes the selected expressions from the Expressions view.

Remove all expressions

This command removes all expressions from the *Expressions* view.

Change variable value

This command allows you to change the value of the selected variable.

Show constants

This option can be toggled to show or hide constants (static final fields).

Show static fields

This option can be toggled to show or hide static fields.

Show qualified names

This option can be toggled to show or hide qualified names.

Show type names

This option can be toggled to show or hide type names.

Add/Remove watchpoint

This command allows you to add or remove a field watchpoint for the current selected variable in the *Expressions* view.

Inspect

This command causes the selected variables to be inspected.

Open declared type

This command allows you open an editor on the declared type of the currently selected variable in the *Expressions view*

Show qualified names

This option can be toggled to show or hide qualified names.

Show type names

This option can be toggled to show or hide type names.

Add/Remove watchpoint

This command allows you to add or remove a field watchpoint for the current selected variable in the *Variables* view.

Change variable value

This command allows you to change the value of the selected variable.

Inspect

This command causes the selected variable(s) to be inspected.

Step commands

Many of the commands in this menu allow you to step through code being debugged. [See Debug View](#)

JUnit

This page lets you configure the stack trace filter patterns for JUnit. The patterns are used in the *JUnit* view to control which entries are visible in stack traces.

Action	Description
Add Filter...	Allows to add a custom filter pattern. This action inserts an empty pattern which can be edited.
Add Class...	Allows to add classes to be filtered. This action opens the <i>Open Type</i> dialog to choose a type to be filtered in the stack traces.
Add Packages...	Allows to add packages to be filtered. This action opens a package selection dialog to choose the packages to be filtered in the stack traces.
Remove	Removes the currently selected stack trace filter pattern.
Enable All	Enables all stack trace filter patterns.
Disable All	Disables all stack trace filter patterns.

■ Related tasks

Using JUnit

■ Related reference

Open type

Java Task Tags page

The options in this page indicate the task tags for a Java project.
You can reach this page through the

- Java task tags property page (File > Properties > Java Compiler > Task Tags) from the context menu on a created project or the [File menu](#)

A project can either reuse workspace default settings or use its own custom settings.

Option	Description
Enable project specific settings	Once selected, task tags can be configured for this project as in the Task tags preference page . At any time, it is possible to revert to workspace defaults, by using the button Restore Defaults .

Related reference

[Task tag preferences](#)

Java Build Path page

The options in this page indicate the build path settings for a Java project. You can reach this page through the [New Java Project wizard](#).






The build class path is a list of paths visible to the compiler when building the project.

Source tab



Source folders are the root of packages containing .java files. The compiler will translate the contained files to .class files that will be written to the output folder. The output folder is defined per project except if a source folder specifies an own output folder. Each source folder can define an exclusion filter to specify which resources inside the folder should not be visible to the compiler.

Resources existing in source folders are also copied to the output folder unless the setting in the [Compiler preference page](#) (Window > Preferences > Java > Compiler > Building) specifies that the resource is filtered.

The tree shows the project as it will look like when switching to the package explorer. Several operations can be executed on this tree to change the structure of the project.

Icon	Option	Description
	Add to build path	Allows to add a folder or package to the Java build path and change it to a source folder. A source folder is a top-level folder in the project hierarchy that is used as the root of packages. Entries on the build path like source folders are visible to the compiler and all contained resources like .java files are used to build the project. Source folders allow to structure the project, for example to separate test from the application in two source folders. Within a source folder, a more detailed structuring can be done by using packages.
	Remove from buildpath	Allows to remove a source folder from the Java build path and change it into a normal folder. All contained resources in this folder (like .java files) are no longer visible to the compiler and will not be included when building the project.
	Exclude	Allows to add a resource to the exclusion filter of its parent source folder. The consequence is that all children of this resource are no longer visible to the compiler. This operation can be useful if for example, some packages are not needed in the project and can therefore be hidden from the compiler.
	Include	This action is available on files or folders that have been excluded. In this situation, include allows to make these resources again visible to the compiler.
	Configure source folder properties	Editing can be used in two situations: <ol style="list-style-type: none">1. Customize the inclusion and exclusion filters by defining string patterns. This can be useful if including or excluding every single resource would take too long and just some simple patterns can do the job. A very practical operator is the wildcard operator to define more sophisticated patterns (for example exclude all resources that start with "Test*.java").2. Change the output folder for a source folder. The consequence is that all generated .class files from the .java files in this source folder will be

Basic tutorial

		generated in the separate output folder instead of the project's output folder. Note that this advanced action is only available on single selected objects.
	Undo all changes	All changes that have been applied to the project in this wizard will be withdrawn and the original state of the project is reconstructed.
	Link additional source to project	When creating a project, there are might already exist some pieces of code or other resources somewhere in the file system that could also be used for this new project. To add this sources to the project, it is possible to create a linked source folder to the folder in the file system and make its content visible to the compiler. Note that only a link to the folder is created. This means that any modifications on resources in that folder will also have an impact on other projects also using this resources.
No icon	Allow output folders for source folders	Shows/Hides the 'output folder' attribute of the source folders. If no output folders are shown, this means that the project's default output folder is used for the generated .class files.

Note that a shorter description of all operations is visible in the area at the bottom of the project tree (labeled with 'Description'). Only the descriptions which are valid for the current selection on the project are shown. For experienced users it is also possible to close the description area to view the projects structure enlarged.

Projects tab

In the **Required projects on the build path** list, you can add project dependencies by selecting other workbench projects to add to the build path for this new project. The **Select All** and **Deselect All** buttons can be used to add or remove all other projects to or from the build path.

Adding a required project indirectly adds all its classpath entries marked as 'exported'. Setting a classpath entry as exported is done in the Order and Export tab.

The projects selected here are automatically added to the referenced projects list. The referenced project list is used to determine the build order. A project is always build after all its referenced projects are built.

Libraries tab

On this page, you can add libraries to the build path. You can add:

- Workbench–managed (internal) JAR files
- File system (external) JAR files
- Folders containing CLASS files
- Predefined libraries like the JRE System Library

JAR files can also be added indirectly as class path variables.

By default, the library list contains an entry representing the Java runtime library. This entry points to the JRE selected as the default JRE. The default JRE is configured in the [Installed JREs preferences page](#) (Window > Preferences > Java > Installed JREs)

Libraries tab options

Option	Description
Add JARs	Allows you to navigate the workbench hierarchy and select JAR files to add to the build path.
Add External JARs	Allows you to navigate the file system (outside the workbench) and select JAR files to add to the build path.
Add Variable	Allows you to add classpath variables to the build path. Classpath variables are an indirection to JARs with the benefit of avoiding local file system paths in a classpath. This is needed when projects are shared in a team. Variables can be created and edited in the Classpath Variable preference page (Window > Preferences > Java > Build Path > Classpath Variables)
Add Library	Allows to add a predefined libraries like the JRE System Library. Such libraries can stand for an arbitrary number of entries (visible as children node of the library node)
Add Class Folder	Allows to navigate the workbench hierarchy and select a class folder for the build path. The selection dialog also allows you to create a new folder.
Edit	Allows you to modify the currently selected library entry or entry attribute
Remove	Removes the selected element from the build path. This does not delete the resource.

Libraries have the following attributes (presented as library entry children nodes):

Library entry attributes

Attribute	Description
Javadoc location	Specifies where the library's Javadoc documentation can be found. If specified you can use Shift+F2 on an element of this library to open its documentation.
Source attachment	Specifies where the library's source can be found.

Order and Export tab

In the **Build class path order** list, you can click the **Up** and **Down** buttons to move the selected path entry up or down in the build path order for this new project.

Checked list entries are marked as exported. Exported entries are visible to projects that require the project. Use the **Select All** and **Deselect All** to change the checked state of all entries. Source folders are always exported, and can not be deselected.

Default output folder

At the bottom of this page, the **Default output folder** field allows you to enter a path to a folder path where the compilation output for this project will reside. The default output is used for source folders that do not specify an own output folder. Use **Browse** to select an existing location from the current project.

Related concepts

[Build classpath](#)

[Classpath variables](#)

■ Related tasks

[Working with build paths](#)

[Attaching source to variables](#)

[Attaching source to a JAR file](#)

■ Related reference

[Frequently asked questions on JDT](#)

[Classpath Variables preferences](#)

[Java Compiler properties](#)

Refactoring

The goal of refactoring support is to allow for improving your code without changing its behavior. When you refactor your code, your goal is to make a system-wide coding change without affecting the semantic behavior of the system. The JDT automatically manages refactorings for you.

The workbench optionally allows you to preview all the impending results of a refactoring action before you finally choose to carry it out.

Refactoring commands are available from the context menus in many views and editors and the Refactor menu in the menu bar.

■ [Related concepts](#)

[Refactoring support](#)

■ [Related reference](#)

[Refactoring actions](#)

[Refactoring wizard](#)

Tips and Tricks

[Editing](#)

[Searching](#)

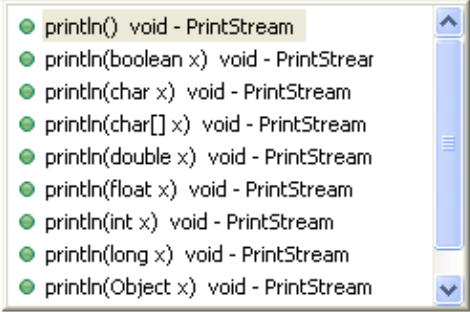
[Navigation](#)

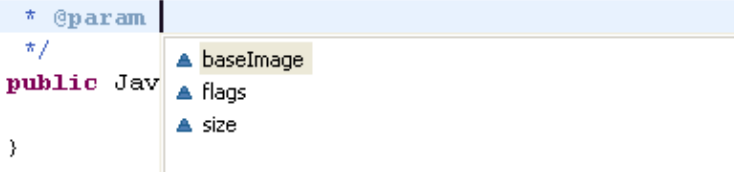
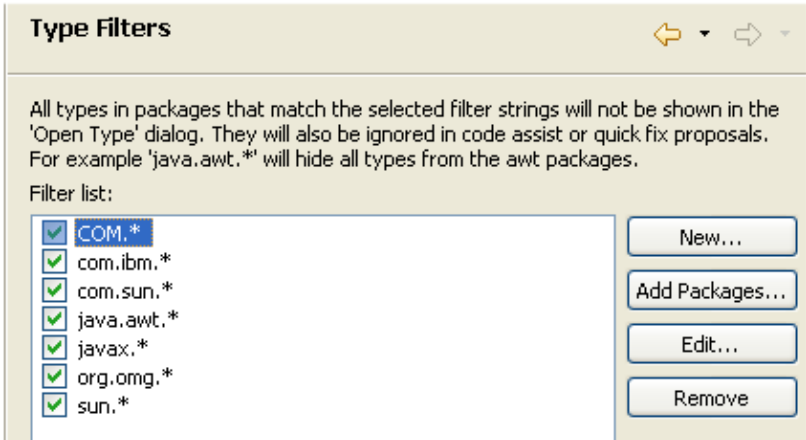
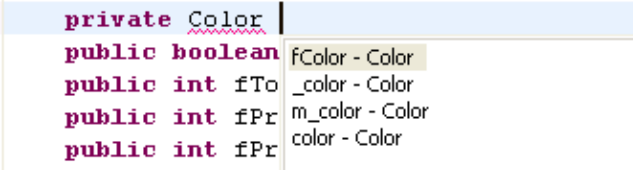
[Views](#)

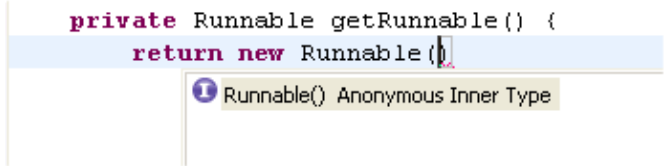
[Debugging](#)

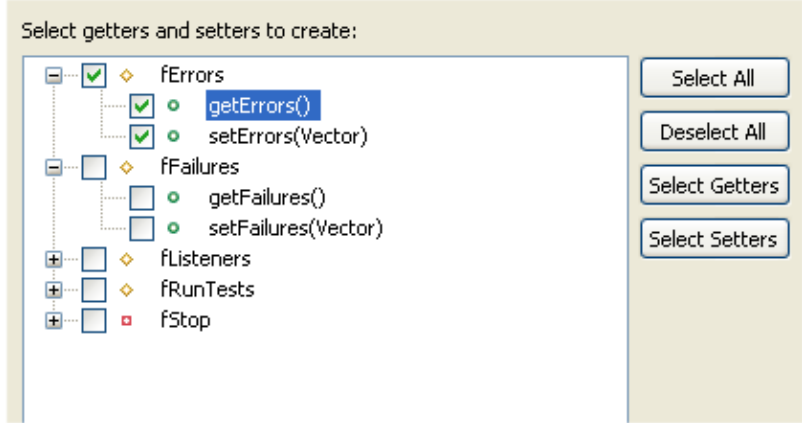
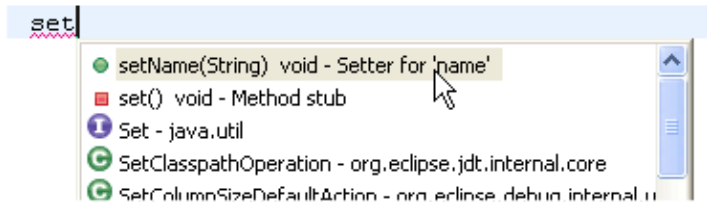
[Miscellaneous](#)

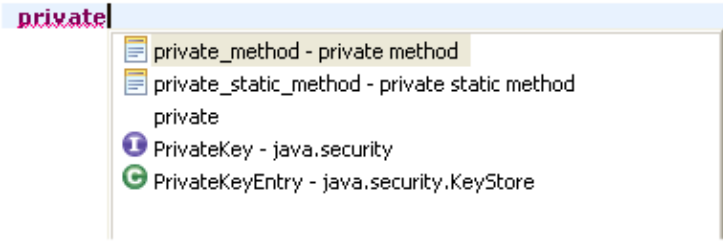
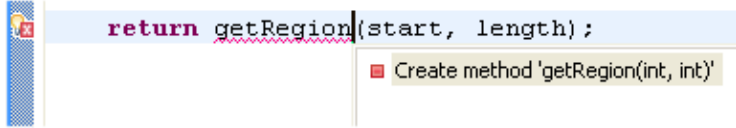
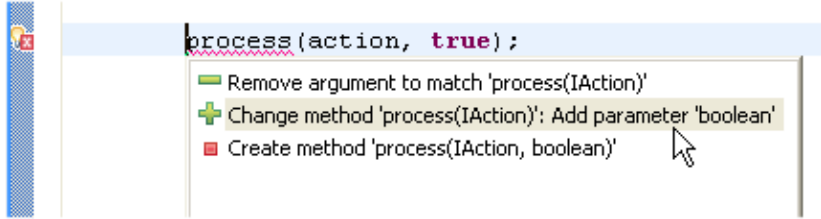
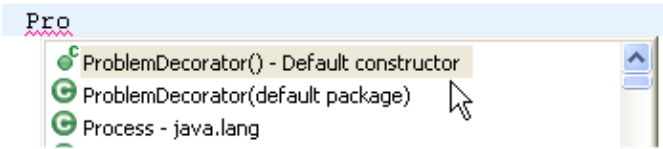
Editing source

Content assist	<p>Content assist provides you with a list of suggested completions for partially entered strings. In the Java editor press Ctrl+Space or invoke Edit > Content Assist.</p> <pre>public class Main { public static void main(String[] args) { System.out.println } }</pre>  <p>The screenshot shows a Java Content Assist popup window. The popup lists several <code>println()</code> methods from the <code>PrintStream</code> class, each preceded by a green circular icon. The methods listed are: <code>println()</code>, <code>println(boolean x)</code>, <code>println(char x)</code>, <code>println(char[] x)</code>, <code>println(double x)</code>, <code>println(float x)</code>, <code>println(int x)</code>, <code>println(long x)</code>, and <code>println(Object x)</code>. The first method, <code>println()</code>, is highlighted with a yellow background.</p>
-----------------------	--

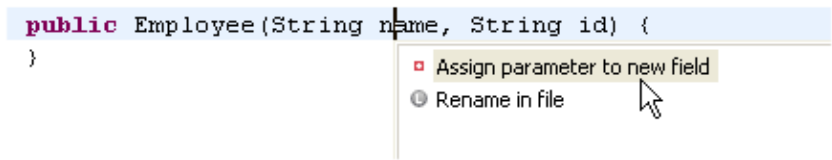
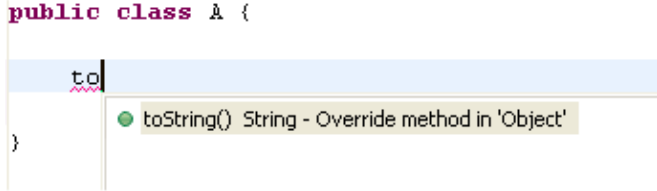
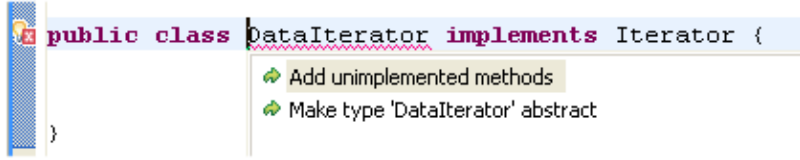
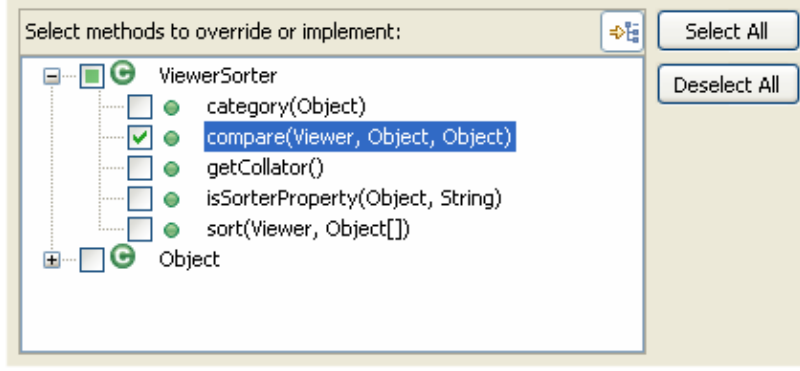
<p>Content assist in Javadoc comments</p>	<p>Content assist is also available in Javadoc comments.</p> <pre> /** * Creates a new JavaElementImageDescriptor. * * @param baseImage an image descriptor used as * @param flags flags indicating which adornment * for valid values. * @param */ public Java </pre> 
<p>Suppress types in code assist</p>	<p>To exclude certain types from appearing in content assist, use the type filter feature configured on the Window > Preferences > Java > Appearance > Type Filters preference page. Types matching one of these filter patterns will not appear in the Open Type dialog and will not be available to code assist, quick fix and organize imports. These filter patterns do not affect the Package Explorer and Type Hierarchy views.</p> 
<p>Content assist for variable, method parameter and field name completions</p>	<p>You can use content assist to speed up the creation of fields, method parameters and local variables. With the cursor positioned after the type name of the declaration, invoke Edit > Content Assist or press Ctrl+Space.</p> <pre> public class ProgressBar extends Canvas { private Color public boolean public int fTo public int fPr public int fPr </pre>  <p>If you use a name prefix or suffix for fields, local variables or method parameters, be sure to specify this in the Code Style preference page (Window > Preferences > Java > Code Style).</p>

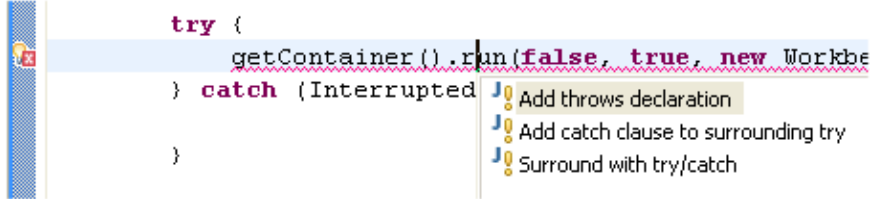
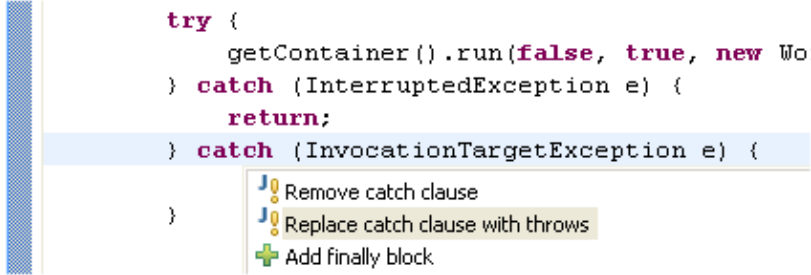
Parameter Hints	<p>With the cursor in a method argument, you can see a list of parameter hints. In the Java Editor press Ctrl+Shift+Space or invoke Edit > Parameter Hints.</p> <pre> if (moveCursor) { int selectionOffset, int selectionLength setSelectedRange(start, 0); revealRange(start, length); } </pre>
Content assist on anonymous classes	<p>Content assist also provides help when creating an anonymous class. With the cursor positioned after the opening bracket of a class instance creation, invoke Edit > Content Assist or press Ctrl+Space.</p> <pre> private Runnable getRunnable() { return new Runnable() { </pre>  <p>This will create the body of the anonymous inner class including all methods that need to be implemented.</p>
Toggle between inserting and replacing code assist	<p>When code assist is invoked on an existing identifier, code assist can either replace the identifier with the chosen completion or do an insert. The default behavior (overwrite or insert) is defined in Window > Preferences > Java > Editor > Code Assist.</p> <p>You can temporarily toggle the behavior while inside the content assist selection dialog by pressing and holding the Ctrl key while selecting the completion.</p>
Incremental content assist	<p>Per default, content assist will now Insert common prefixes automatically, similar to Unix shell expansion. To change that behavior uncheck the setting on the Window > Preferences > Java > Editor > Code Assist preference page.</p>
Create Getter and Setters dialog	<p>To create getter and setter methods for a field, select the field's declaration and invoke Source > Generate Getter and Setter.</p>

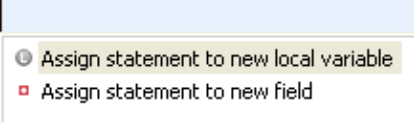
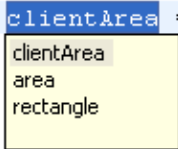
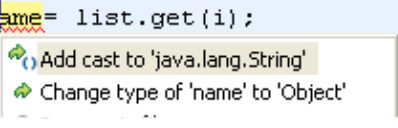
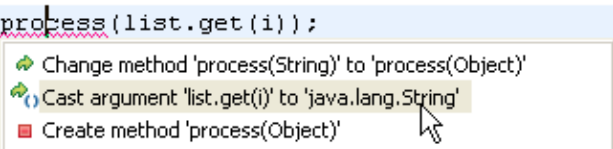
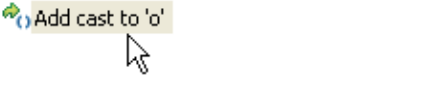
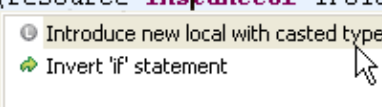
	 <p>Select getters and setters to create:</p> <p>If you use a name prefix or suffix be sure to specify this in the Code Style preference page (Window > Preferences > Java > Code Style).</p>
<p>Use content assist to create Getter and Setters</p>	<p>Another way to create getters and setters is using content assist. Set the cursor in the type body between members and press Ctrl+Space to get the proposals that create a getter or setter method stub.</p> 
<p>Delete Getters and Setters together with a field</p>	<p>When you delete a field from within a view, Eclipse can propose deleting its Getter and Setter methods. If you use a name prefix or suffix for fields, be sure to specify this in the Code Style preference page (Window > Preferences > Java > Code Style).</p>
<p>Create delegate methods</p>	<p>To create a delegate method for a field select the field's declaration and invoke Source > Generate Delegate Methods. This adds the selected methods to the type that contains a forward call to delegated methods. This is an example of a delegate method:</p> <pre>public void addModifyListener(ModifyListener listener) { fTextControl.addModifyListener(listener); }</pre>
<p>Use Drag & Drop for refactoring</p>	<p>You can move Java compilation units between packages by Drag & Drop – all missing imports will be added and references updated.</p>
<p>Use Drag & Drop to move and copy Java code elements</p>	<p>You can move and copy Java elements such as methods and fields by Drag & Drop. This will not trigger refactoring – only the code will be copied or moved.</p>


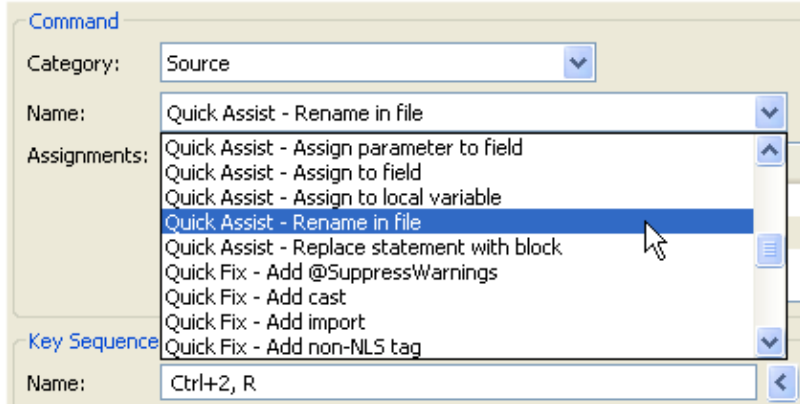
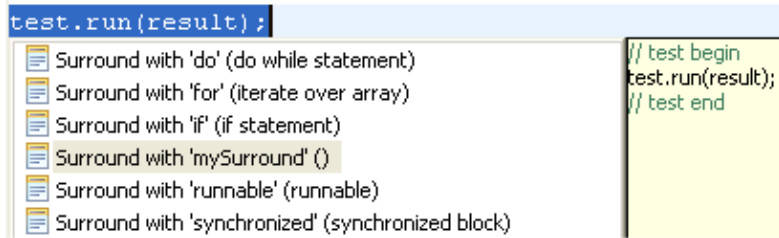
<p>Use Templates to create a method</p>	<p>You can define a new template (Window > Preferences > Java > Editor > Templates) that contains a method stub. Templates are shown together with the Content Assist (Ctrl+Space) proposals.</p> <p>There are also existing templates, such as 'private_method', 'public_method', 'protected_method' and more.</p> <p>Use the Tab key to navigate between the values to enter (return type, name and arguments).</p> 
<p>Use Quick Fix to create a new method</p>	<p>Start with the method invocation and use Quick Fix (Ctrl+1) to create the method.</p> 
<p>Use Quick Fix to change a method signature</p>	<p>Add an argument to a method invocation at a call site. Then use Quick Fix (Ctrl+1) to add the required parameter in the method declaration.</p> 
<p>Use Content Assist to create a constructor stub</p>	<p>At the location where you want to add the new constructor, use code assist after typing the first letters of the constructor name.</p> 
<p>Create new fields from parameters</p>	<p>Do you need to create new fields to store the arguments passed in the constructor? Use Quick Assist (Ctrl+1) on a parameter to create the assignment and the field declaration and let Eclipse propose a name according to your Code Style preferences.</p>

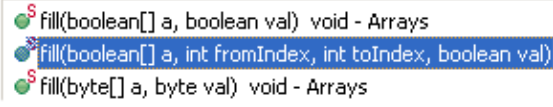
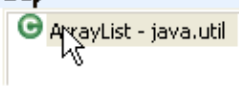
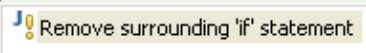
Basic tutorial

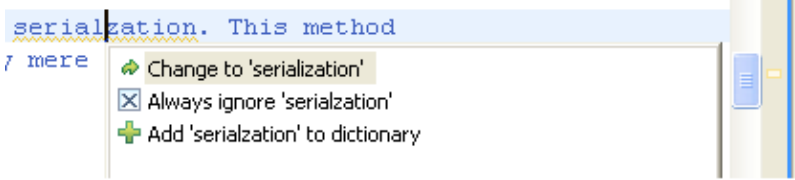
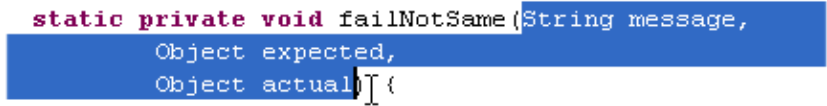
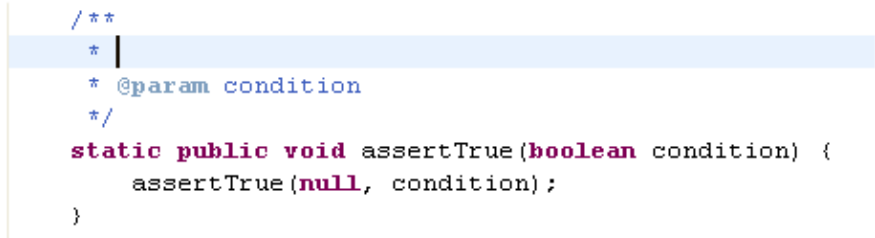
	
Use Content Assist to override a method	<p>Invoke Content Assist (Ctrl+Space) in the type body at the location where the method should be added. Content assist will offer all methods that can be overridden. A method body for the chosen method will be created.</p> 
Use Quick Fix to add unimplemented methods	<p>To implement a new interface, add the 'implements' declaration first to the type. Even without saving or building, the Java editor will underline the type to signal that methods are missing and will show the Quick Fix light bulb. Click on the light bulb or press Ctrl+1 (Edit > Quick Fix) to choose between adding the unimplemented methods or making your class abstract.</p> 
Override a method from a base class	<p>To create a method that overrides a method from a base class: Select the type where the methods should be added and invoke Source > Override / Implement Methods. This opens a dialog that lets you choose which methods to override.</p> 

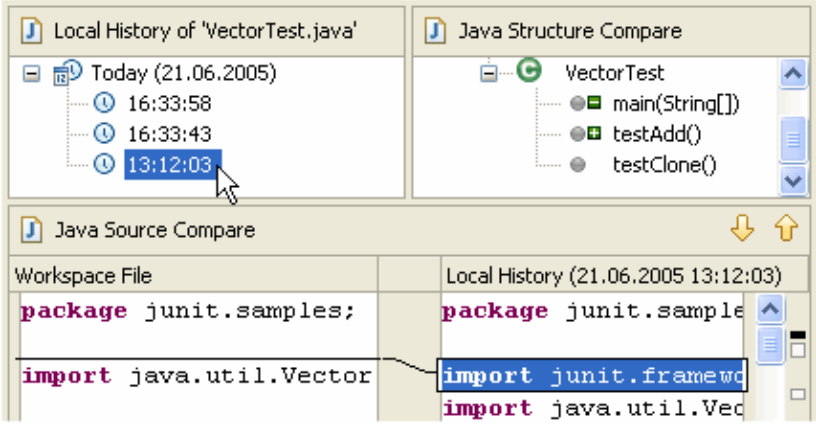
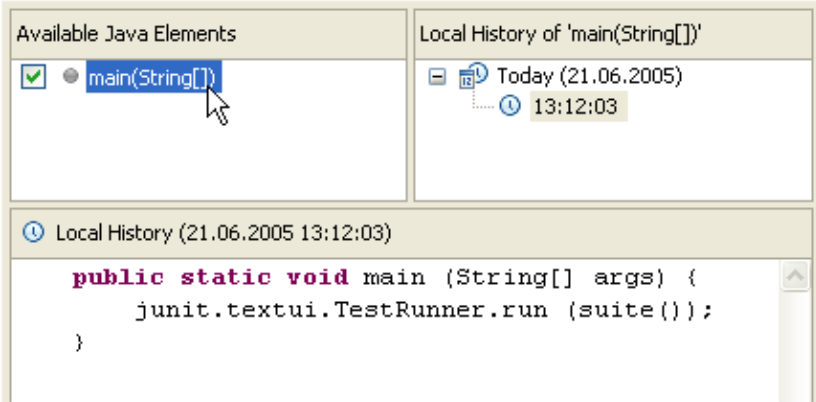
Rename in File	<p>To quickly do a rename that doesn't require full analysis of dependencies in other files, use the 'rename in file' Quick Assist. In the Java Editor, position the cursor in an identifier of a variable, method or type and press Ctrl+1 (Edit > Quick Fix)</p> <p>The editor is switched to the linked edit mode (like templates) and changing the identifier simultaneously changes all other references to that variable, method or type.</p> <pre> public void run(ActivityResult result) { for (Enumeration e= tests(); e.hasMoreElements();) { if (result.shouldStop()) break; Test test= (Test)e.nextElement(); runTest(test, result); } } </pre>
Use Quick Fix to handle exceptions	<p>Dealing with thrown exceptions is easy. Unhandled exceptions are detected while typing and marked with a red line in the editor.</p> <ul style="list-style-type: none"> • Click on the light bulb or press Ctrl+1 to surround the call with a try catch block. If you want to include more statements in the try block, select the statements and use Source > Surround With try/catch Block. You can also select individual statements by using Edit > Expand Selection to and selecting Enclosing, Next or Previous. • If the call is already surrounded with a try block, Quick Fix will suggest adding the catch block to the existing block. • If you don't want to handle the exception, let Quick Fix add a new thrown exception to the enclosing method declaration  <p>At any time you can convert a catch block to a thrown exception. Use Ctrl+1 (Edit > Quick Fix) on a catch block.</p> 

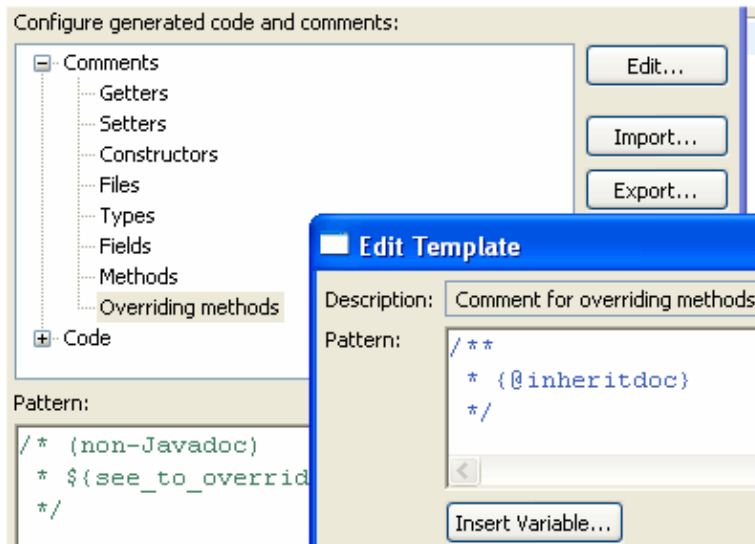
<p>Less typing for assignments</p>	<p>Instead of typing an assignment, start with the expression that will be assigned.</p> <pre>textControl.getClientArea();</pre>  <p>Now use Ctrl+1 (Edit > Quick Fix) and choose 'Assign statement to new local variable' and Quick Assist will guess a variable name for you.</p> <pre>Rectangle clientArea = textControl.getClientArea();</pre> 
<p>Less work with cast expressions</p>	<p>Don't spend too much time with typing casts. Ignore them first and use quick assist to add them after finishing the statement.</p> <p>For example on assignments:</p> <pre>String name = list.get(i);</pre>  <p>Or in for method arguments:</p> <pre>process(list.get(i));</pre>  <p>Or for method call targets</p> <pre>if (o instanceof Runnable) { o.run(); }</pre> 
<p>Assign a casted expression</p> <p>3.1</p>	<p>After an 'instanceof' check, it is very common to cast the expression and assign it to a new local variable. Invoke Quick Assist (Ctrl+1) on the 'instanceof' keyword to save yourself some typing:</p> <pre>if (resource instanceof IFolder) {</pre>  <pre> IFolder folder = (IFolder) resource; }</pre>
<p>More quick assists</p> <p>3.1</p>	<p>Check out the quick assist page for a complete list of available code transformations. Amongst them are</p> <ul style="list-style-type: none"> • Invert if statements

	<ul style="list-style-type: none"> • Convert 'switch' into 'if-else' • Replace 'if-else' with conditional ...and many more <p>A list of quick fixes can be found here.</p>
Shortcuts for Quick Fixes and Assists 	<p>Some of the popular quick assists like Rename In File and Assign To Local can be invoked directly with Ctrl+2 R and Ctrl+2 L. Check the keys preference page for more quick fixes that support direct invocation.</p> 
Surround lines	<p>To surround statements with an if / while / for statement or a block, select the lines to surround and press Ctrl+1 (Edit > Quick Fix). This lists all templates that contain the variable <code>\${line_selection}</code>.</p>  <p>Templates can be configured on Window > Preferences > Java > Editor > Templates. Edit the corresponding templates or define your own templates to customize the resulting code.</p>
Create your own templates	<p>To create your own templates, go to the Window > Preferences > Java > Editor > Templates preference page and press the New button to create a template. For example, a template to iterate backwards in an array would look like this:</p> <pre>for (int \${index}= \${array}.length - 1; \${index} >= 0; \${index}--){ \${cursor} }</pre>

<p>Code assist can insert argument names automatically</p>	<p>You can have code assist insert argument names automatically on method completion. This behavior can be customized on the Window > Preferences > Java > Editor > Code Assist preference page (see the Fill argument names on completion checkbox.) For example, when you select the second entry here,</p> <pre>java.util.Arrays.fill</pre>  <p>code assist will automatically insert argument names:</p> <pre>java.util.Arrays.fill(boolean[] a, int fromIndex, int toIndex, boolean val)</pre> <p>you can then use the Tab key to navigate between the inserted names.</p> <p>Code assist can also guess argument names – based on their declared types. This can be configured by the Guess filled argument names checkbox on the Window > Preferences > Java > Editor > Code Assist preference page.</p>
<p>Automatically insert type arguments</p> <p>3.1</p>	<p>Enabling Fill argument names on completion on the Window > Preferences > Java > Editor > Code Assist preference page is also useful when working with parameterized types in J2SE 5.0.</p> <pre>List<String> strings= new ArrayList</pre>  <p>results in</p> <pre>List<String> strings= new ArrayList<String>();</pre>
<p>Remove surrounding statement</p>	<p>To remove a surrounding statement or block, position the cursor at the opening bracket and press Ctrl+1 (Edit > Quick Fix).</p> <pre>if (text.length() > 0) { return text; }</pre> 

<p>How was that word spelled again?</p>	<p>You can enable spell-checking support in the Java editor on the General > Editors > Text Editors > Spelling preference page. Spelling errors are displayed in the Java editor and corresponding Quick Fixes are available:</p>  <p>You can make the dictionary also available to the content assist. However, there is currently no dictionary included in Eclipse. The required format is just a list of words separated by new lines and the Quick Fixes allow you to add new words to the dictionary on-the-fly. Contributions of dictionaries would be welcome.</p>
<p>Structured selections</p>	<p>You can quickly select Java code syntactically using the Structured Selection feature. Highlight the text and press Alt+Shift+Arrow Up or select Edit > Expands Selection To > Enclosing Element from the menu bar – the selection will be expanded to the smallest Java-syntax element that contains the selection. You can then further expand the selection by invoking the action again.</p>
<p>Find the matching bracket</p>	<p>To find a matching bracket select an opening or closing bracket and press Ctrl+Shift+P (Navigate > Go To > Matching Bracket). You can also double click before an opening or after a closing bracket – this selects the text between the two brackets.</p> 
<p>Smart Javadoc</p>	<p>Type /** and press Enter. This automatically adds a Javadoc comment stub containing the standard @param, @return and @exception tags.</p>  <p>The templates for the new comment can be configured in Window > Preferences > Java > Code Style > Code Templates</p>

<p>Use the local history to revert back to a previous edition of a method</p>	<p>Whenever you edit a file, its previous contents are kept in the local history. Java tooling makes the local history available for Java elements, so you can revert back to a previous edition of a single method instead of the full file.</p> <p>Select an element and use Replace With > Local History to revert back to a previous edition of the element.</p> 
<p>Use the local history to restore removed methods</p>	<p>Whenever you edit a file, its previous contents are kept in the local history. Java tooling makes the local history available for Java elements, so you can restore deleted methods selectively.</p> <p>Select a container and use Restore from Local History to restore any removed members.</p>  <pre> public static void main (String[] args) { junit.textui.TestRunner.run (suite()); } </pre>
<p>Customizable code generation</p>	<p>The Window > Preferences > Java > Code Style > Code Templates preference page allows you to customize generated code and comments in a similar way to normal templates. These code templates are used whenever code is generated.</p>



Since 3.1, it is possible to project specific Code templates, that will also be shared in the team if your project is shared. Open the **Properties** on project to enable project specific settings.

Create comments in your code

Comments can be added explicitly with **Source > Add Comment (Ctrl+Shift+J)** or automatically by various wizards, refactorings or quick fixes.

Configure the comment templates on the **Window > Preferences > Java > Code Style > Code Templates** preference page.

Enable or disable the automatic generation of comments either directly on the wizard (e.g. using '**Generate Comment**' checkbox on the new Java type wizards) or by the '**Automatically add new comments for new methods and types**' checkbox of the **Window > Preferences > Java > Code Style** page.

All these settings can also be configured on a per project basis. Open the **Properties** on project to enable project specific settings.

Sort members

You can **Sort Members** of a Java compilation unit according to a category order defined in the **Window > Preferences > Java > Appearance > Members Sort Order** preference page.

You'll find the action under **Source > Sort Members**

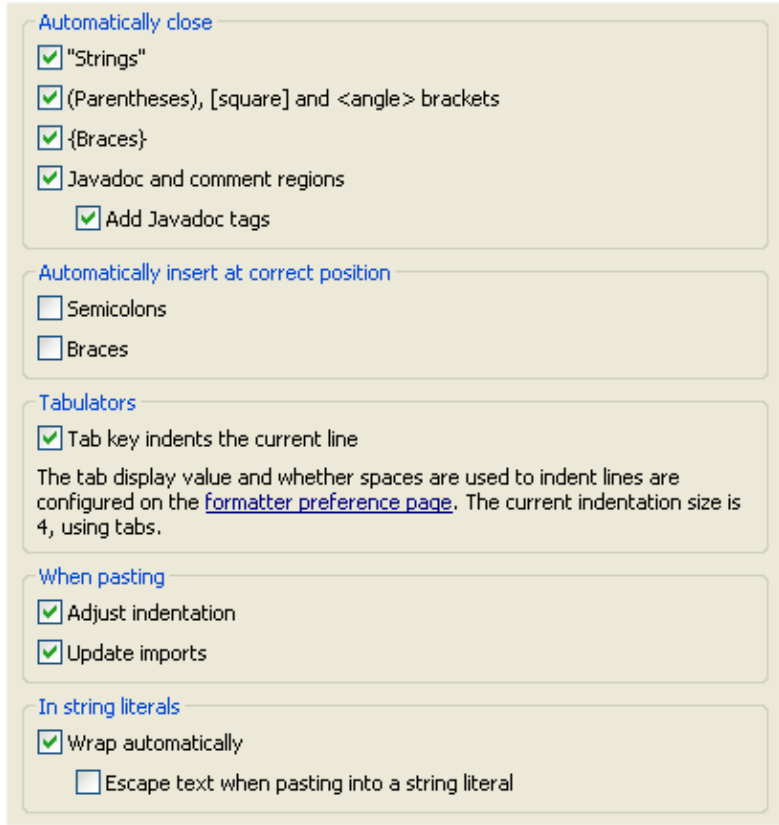
Wrap Strings

You can have String literals wrapped when you edit them. For example, if you have code like this:

```
String message= "This is a very long message.";
```

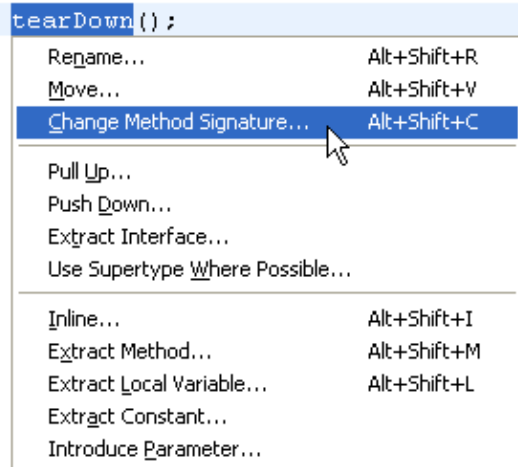
position your caret after the word "very" and press **Enter**. The code will be automatically changed to:

```
String message= "This is a very" +
    " long message.";
```

	<p>This behavior can be customized in the Window > Preferences > Java > Editor > Typing preference page.</p>
<p>Smart Typing and how to control it</p>	<p>The Java editor's Smart Typing features ease your daily work. You can configure them on the Java > Editor > Typing preference page.</p>  <p>When you enable Automatically insert Semicolons at correct position, typing a semicolon automatically positions the cursor at the end of the statement before inserting the semicolon. This saves you some additional cursor navigation.</p> <p>You can undo this automatic positioning by pressing backspace right afterwards.</p>
<p>Fix your code indentation with one key stroke</p>	<p>A useful feature is Source > Correct Indentation or Ctrl+I. Select the code where the indents are incorrect and invoke the action.</p>

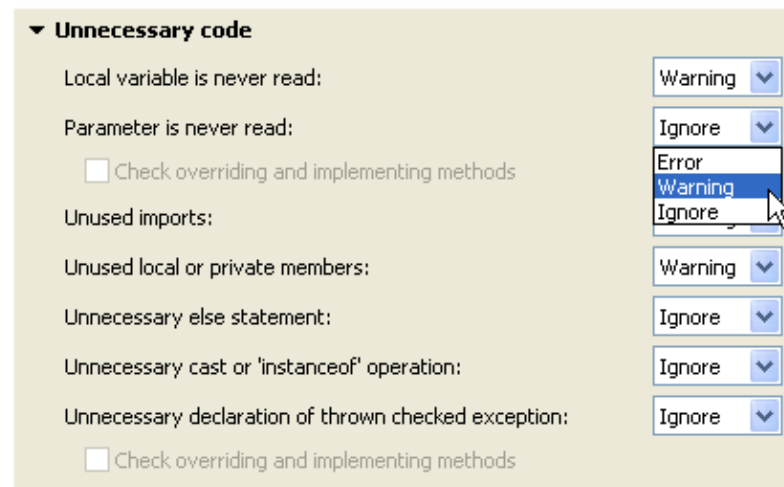
Quick menus for source and refactoring actions

The refactoring and source actions can be accessed via a quick menu. Select the element to be manipulated in the Java editor or in a Java view and press **Alt+Shift+S** for the quick source menu or **Alt+Shift+T** for the quick refactoring menu.

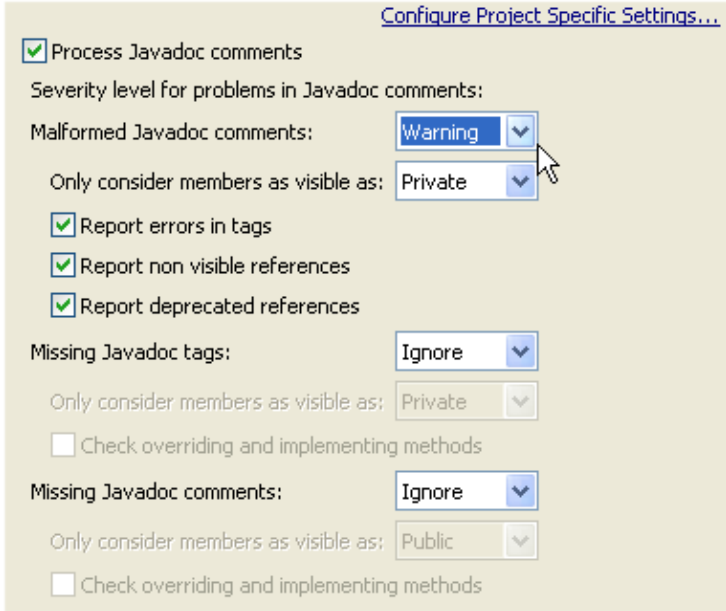
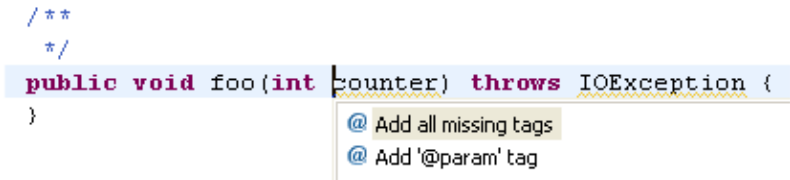

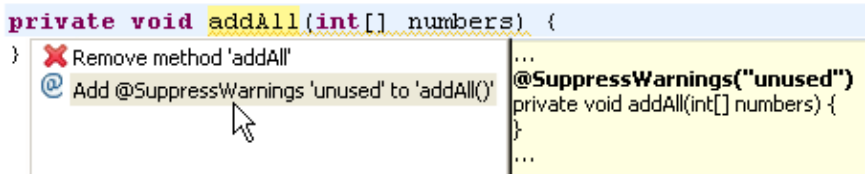


Find unused code

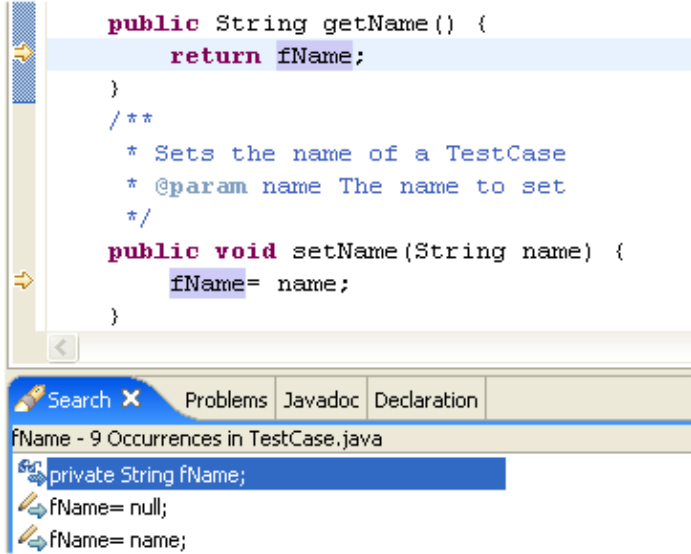
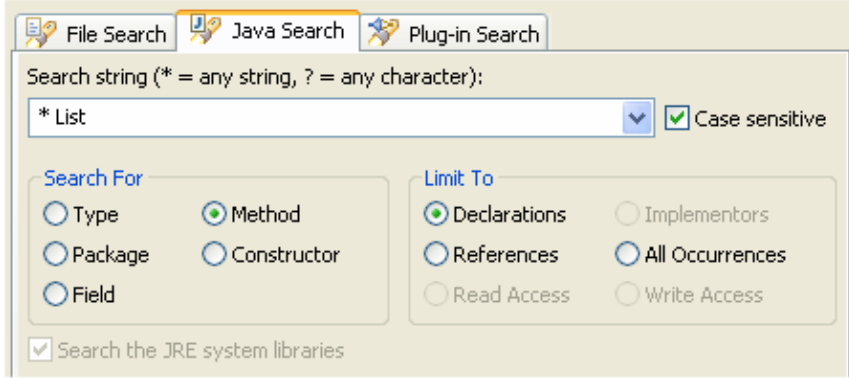
The Java compiler detects unreachable code, unused variables, parameters, imports and unused private types, methods and fields. The setting is on the **Window > Preferences > Java > Compiler > Error/Warnings** preference page (or set for an individual project using **Project > Properties > Java Compiler > Error/Warnings**).

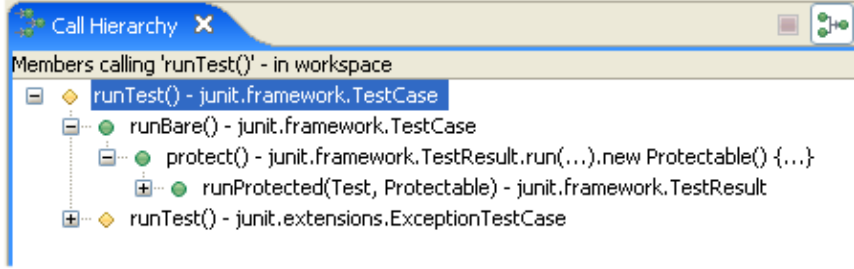


These settings are also detected as you type and a quick fix is offered to remove the unneeded code.

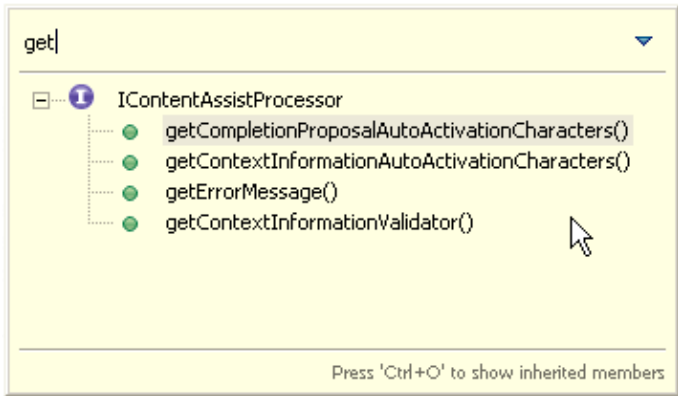
Javadoc comment handling	<p>The Eclipse Java compiler can process Javadoc comments. Search reports references in doc comments, and refactoring updates these references as well. This feature is controlled from the Window > Preferences > Java > Compiler > Javadoc preference page (or set for an individual project using Project > Properties > Java Compiler > Javadoc).</p>  <p>When turned on, malformed Javadoc comments are marked in the Java editor and can be fixed using Edit > Quick Fix (Ctrl+1):</p> 
Suppress warnings 	<p>In J2SE 5.0 you can suppress all optional compiler warnings using the 'SuppressWarnings' annotation. In this example 'addAll()' is marked as an unused method. Quick Fix (Ctrl+1) is used to add a SuppressWarnings annotation so that the warning will not be shown for this method.</p> 

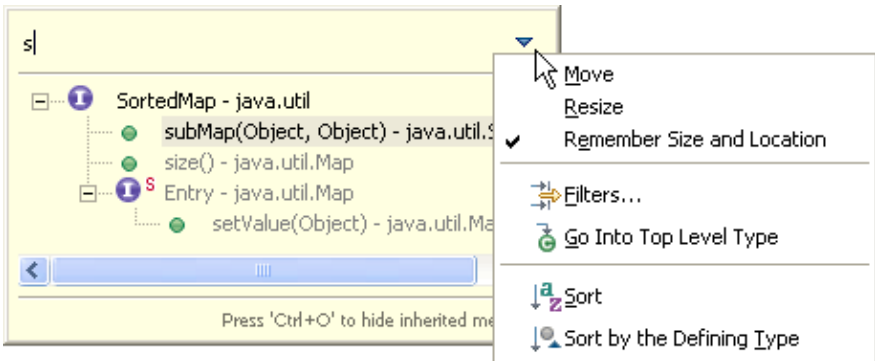
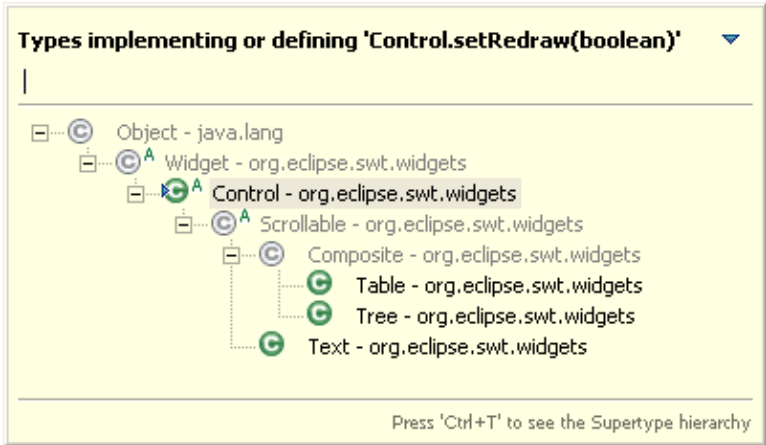
Searching


<p>Locate variables and their read/write access</p>	<p>You can locate variables and see their read/write status by selecting an identifier (variable, method or type reference or declaration) and invoking Search > Occurrences in File. This marks all references of this identifier in the same file. The results are also shown in the search view, along with icons showing the variable's read or write access.</p>  <p>Alternatively, use the Mark Occurrences feature to dynamically highlight occurrences. You can search over several files by using the general search features (Search > References).</p>
<p>Search for methods with a specific return type</p>	<p>To search for methods with a specific return type, use <code>"* <return type>"</code> as follows:</p> <ul style="list-style-type: none"> • Open the search dialog and click on the Java Search tab. • Type '*' and the return type, separated by a space, in the Search string. • Select the Case sensitive checkbox. • Select Method and Declarations and then click Search. 
<p>Remove Javadoc</p>	<p>By default Java Search finds references inside Java code and Javadoc. If you don't want to find references inside Javadoc, you can filter these</p>

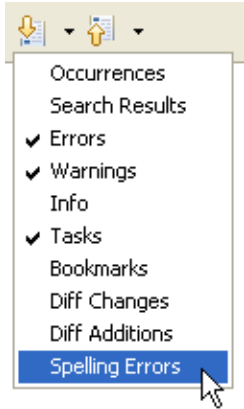
results from Java search	matches by enabling 'Filter Javadoc' in the view menu (triangle symbol) of the search view
Trace method call chains with the Call Hierarchy	<p>Have you ever found yourself searching for references to methods again and again? Use the new Call Hierarchy to follow long or complex call chains without losing the original context: Just select a method and invoke Navigate > Open Call Hierarchy (Ctrl+Alt+H).</p> 

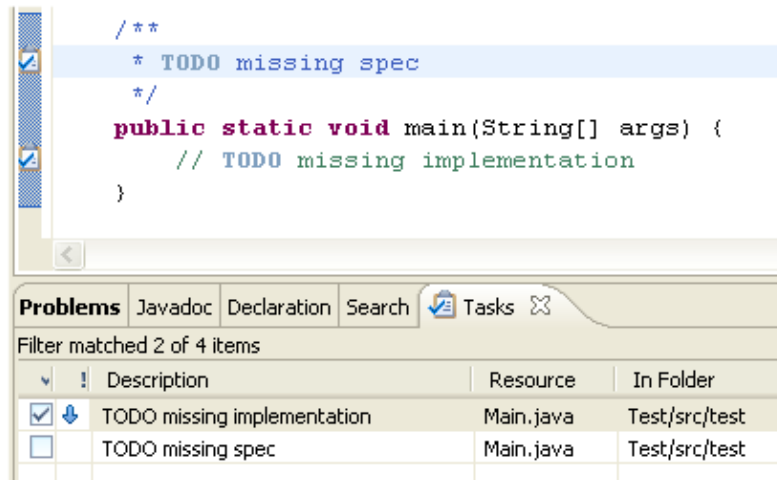
Code navigation and reading

Open on a selection in the Java editor	<p>There are two ways that you can open an element from its reference in the Java editor.</p> <ul style="list-style-type: none"> • Select the reference in the code and press F3 (Navigate > Open Declaration) • Hold Ctrl and move the mouse pointer over the reference <pre> AlertDialog.openError(fShell, title, message); return; </pre> <p>The hyperlink style navigation can be configured in Preferences > General > Editors > Text Editors > Support hyperlink style navigation.</p>
In-place outlines	<p>Press Ctrl+F3 in the Java editor to pop up an in-place outline of the element at the current cursor position. Or press Ctrl+O (Navigate > Quick Outline) to pop up an in-place outline of the current source file.</p> 

<p>In-place outlines show inherited members</p>	<p>Press Ctrl+O or Ctrl+F3 again to add inherited members to an open In-place outline. Inherited members have a gray label. Filter and sort the outline with the menu in the upper right corner.</p> 
<p>In-place hierarchy</p>	<p>Find out which are the possible receivers of a virtual call using the 'quick hierarchy'. Place the cursor inside the method call and press Ctrl+T (Navigate > Quick Hierarchy). The view shows all types that implement the method with a full icon.</p> <pre>control.setRedraw(true);</pre>  <p>Press Ctrl+T again to switch to the Supertype hierarchy.</p>
<p>Advanced highlighting</p>	<p>The Java editor can highlight source code according to its semantics (for example: static fields, local variables, static method invocations). Have a look at the various options on the Window > Preferences > Java > Editor > Syntax Coloring preference page.</p>

	<pre> private static int staticField; /** @deprecated */ private int field; public void foo(int parameter) { int local1= method() + staticMethod(); staticField= local1 + field; } </pre>
Initially folded regions	<p>You can specify which regions are folded by default when an editor is opened. Have a look at the Window > Preferences > Java > Editor > Folding preference page to customize this.</p> <div> <div>Folding</div> <div> <input checked="" type="checkbox"/> Enable folding <div>Initially fold these elements:</div> <div> <input type="checkbox"/> Comments <input checked="" type="checkbox"/> Header Comments <input type="checkbox"/> Inner types <input type="checkbox"/> Methods <input checked="" type="checkbox"/> Imports </div> </div> </div>
Mark occurrences	<p>When working in the editor, turn on Mark Occurrences in the toolbar () or press Alt+Shift+O. You'll see within a file, where a variable, method or type is referenced.</p> <pre> private int getErrorTicksFromMarkers(IResource res) throw int info= 0; IMarker[] markers= res.findMarkers(IMarker.PROBLEM, t if (markers != null) { for (int i= 0; i < markers.length && (info != ERR IMarker curr= markers[i]; int priority= curr.getAttribute(IMarker.SEVER if (priority == IMarker.SEVERITY_WARNING) { info= ERRORTICK_WARNING; } else if (priority == IMarker.SEVERITY_ERROR info= ERRORTICK_ERROR; } } } return info; } </pre> <p>Selecting a return type shows you the method's exit points. Select an exception to see where it is thrown.</p>

	<pre> public static int getVisibilityCode(IBinding binding) { if (isPublic(binding)) return Modifier.PUBLIC; else if (isProtected(binding)) return Modifier.PROTECTED; else if (isPackageVisible(binding)) return Modifier.NONE; else if (isPrivate(binding)) return Modifier.PRIVATE; Assert.isTrue(false); return VISIBILITY_CODE_INVALID; } </pre> <p>Select a super class or interface to see the methods override or implement a method from the selected super type.</p> <p>Fine tune 'mark occurrences' on Window > Preferences > Java > Editor > Mark Occurrences..</p>
Go to next / previous method	<p>To quickly navigate to the next or previous method or field, use Ctrl+Shift+Arrow Up (Navigate > Go To > Previous Member) or Ctrl+Shift+Arrow Down (Navigate > Go To > Next Member)</p>
Control your navigation between annotations	<p>Use the Next / Previous Annotation toolbar buttons or Navigate > Next Annotation (Ctrl+.) and Navigate > Previous Annotation (Ctrl+,) to navigate between annotations in a Java source file. With the button drop-down menus, you can configure on which annotations you want to stop:</p> 
Reminders in your Java code	<p>When you tag a comment in Java source code with "TODO" the Java compiler automatically creates a corresponding task as a reminder. Opening the task navigates you back to the "TODO" in the code. Use the Window > Preferences > Java Compiler > Task Tags preference page to configure any other special tags (like "FIXME") that you'd like to track in the task list.</p>



Tricks in the Open Type dialog

3.1

- To find types quickly, only type the capital letters of the type name: IOOBE finds IndexOutOfBoundsException
- To see all types ending with a given suffix, e.g. all Tests, use '*Test<' to not see all types containing 'Test' somewhere else in the type name.

Select a type to open (? = any character, * = any String, TZ = TimeZone):

Matching types:

IndexOutOfBoundsException - java.lang

Make hovers sticky

You can open the text from a hover in a scrollable window by pressing **F2 (Edit > Show Tooltip Description)**. You can select and copy content from this window.

```
vector.iterator();
```

Iterator java.util.AbstractList.iterator()

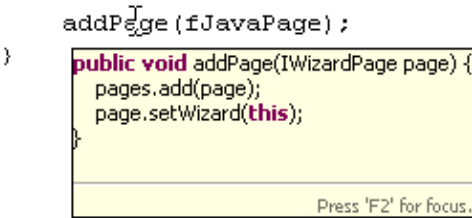

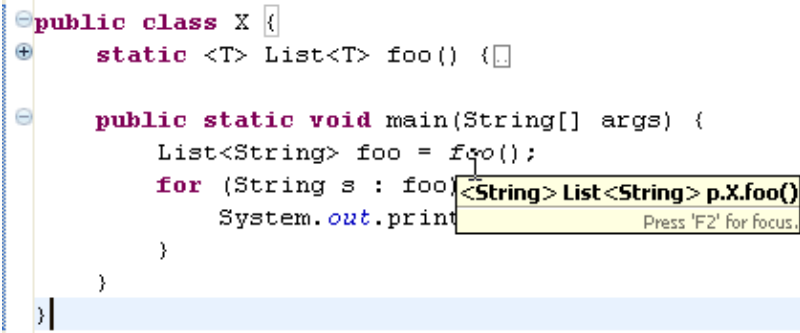
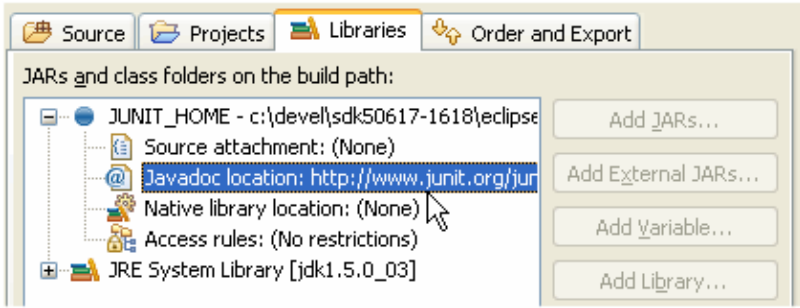
Returns an iterator over the elements in this list in proper sequence. This implementation returns a straightforward implementation of the iterator interface, which implements the methods `hasNext()`, `next()`, `remove()`, `previous()`, `previousIndex()`, `previousElement()`, `nextIndex()`, and `nextElement()`. Note that the iterator returned by this method will throw an `UnsupportedOperationException` if the list's `remove(int)` method is not implemented.

Hovers in the Java editor

You can see different hovers in the Java editor by using the modifier keys (Shift, Ctrl, Alt).

When you move the mouse over an identifier in the Java editor, by default a hover with the Javadoc extracted from the corresponding source of this element is shown. Holding down the Ctrl key shows you the source code.

Basic tutorial

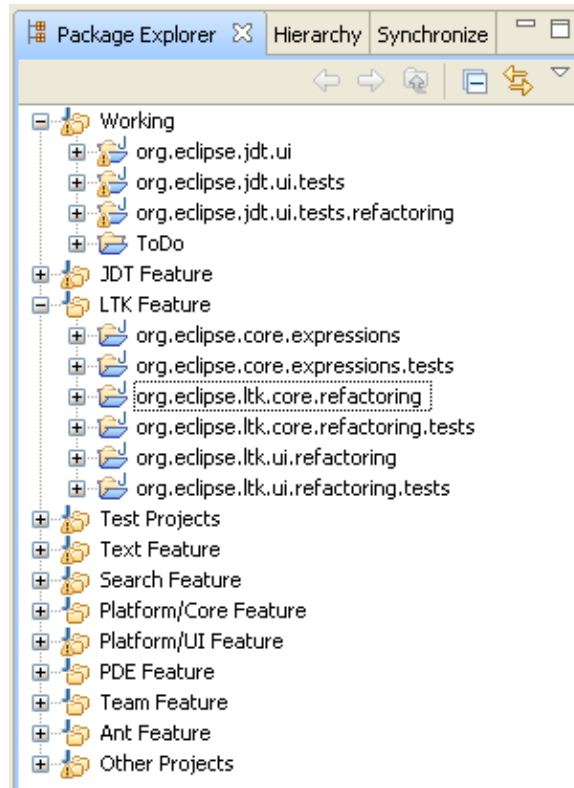
	 <p>You can change this behavior and define the hovers for other modifier keys in Window > Preferences > Java > Editor > Hovers.</p>
Generic method inferred signature 	<p>You can use hover to show the generic method inferred signature.</p> 
Open and configure external Javadoc documentation	<p>If you want to open the Javadoc documentation for a type, method or field with Shift+F2 (Navigate > Open External Javadoc), you first have to specify the documentation locations to the elements parent library (JAR, class folder) or project (source folder).</p> <p>For libraries open the build path page (Project > Properties > Java Build Path), go to the Libraries, expand the node of the library where you can edit the 'Javadoc location' node. The documentation can be local on your file system in a folder or archive or on a web server.</p>  <p>For types, methods or fields in source folders, go to the (Project > Properties > Javadoc Location).</p>

Java views

Organizing workspace with many projects

3.1

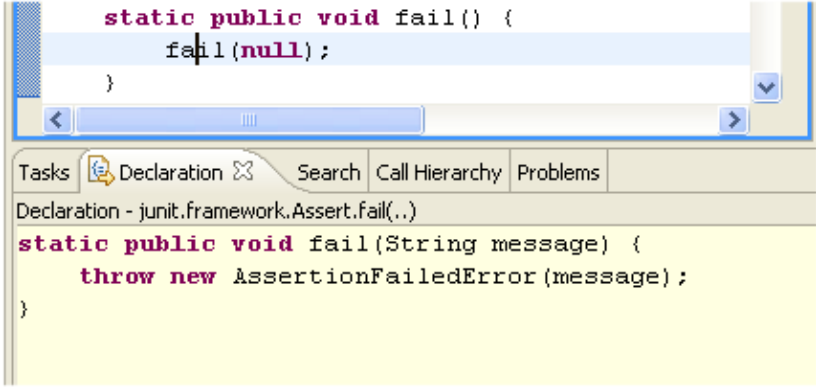
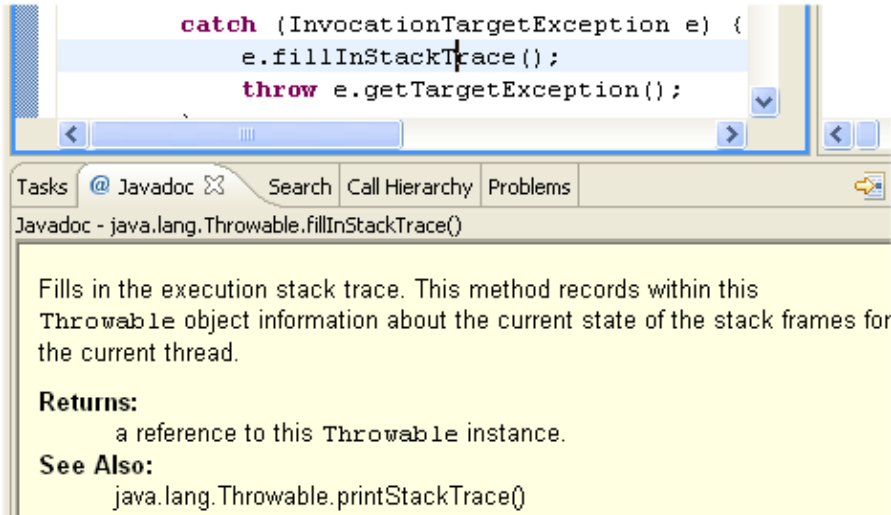
Use **Show > Working Sets** in the Package Explorer's view menu to enable a new mode that shows working sets as top level elements. This mode makes it much easier to manage workspaces containing lots of projects.

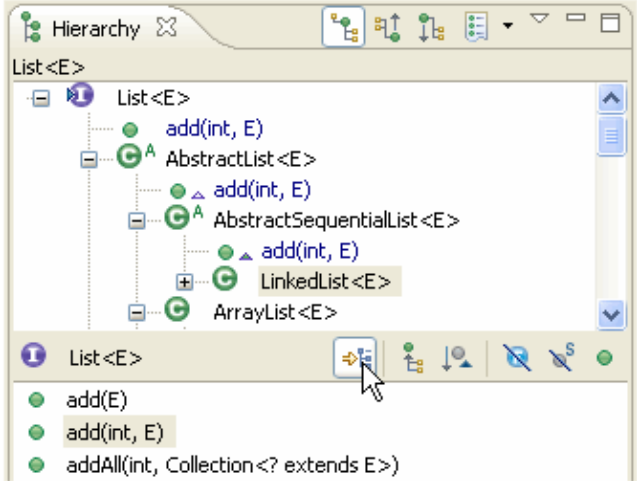
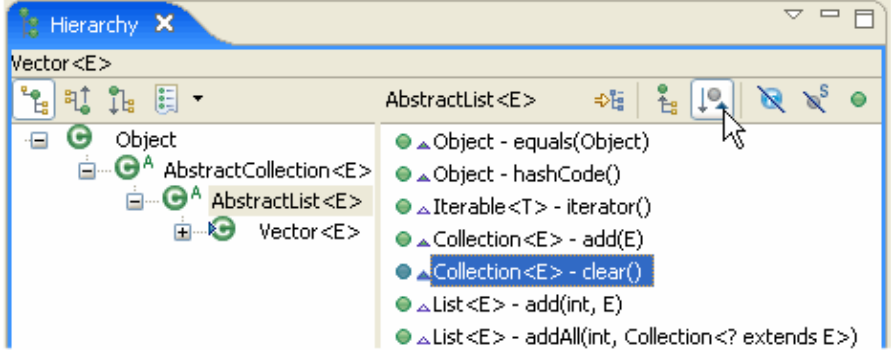


Use **Select Working Sets** from the Package Explorer's view menu to configure which working sets get shown. The dialog lets you create new Java working sets, define which working sets are shown and in what order. Working sets can also be rearranged directly in the Package Explorer using drag and drop and copy/paste.

Declaration view

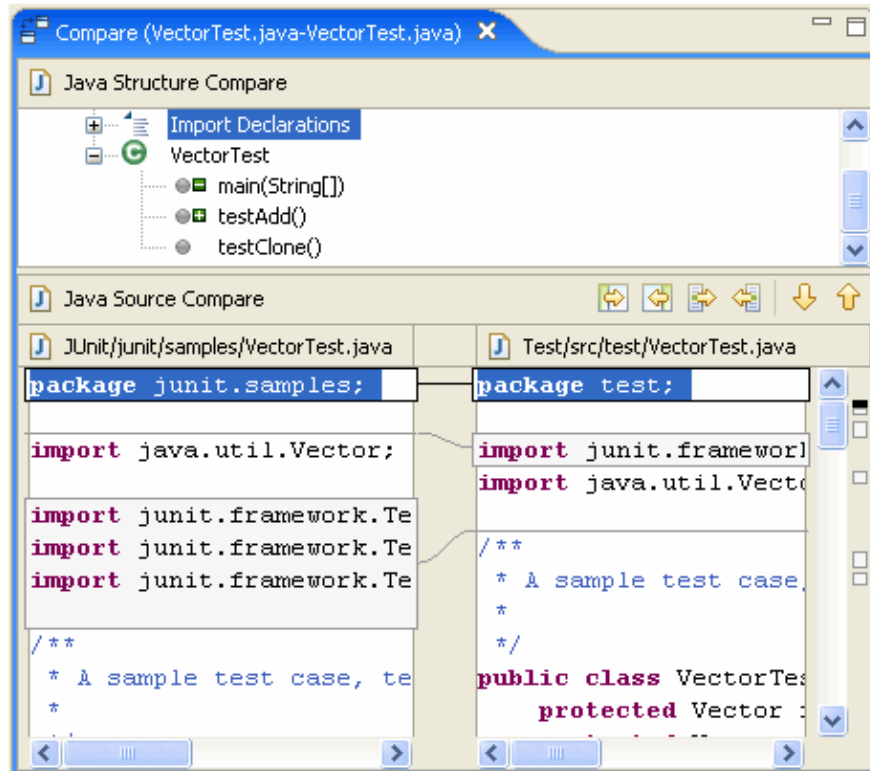
The Declaration view (**Window > Show View > Other > Java > Declaration**) shows the source of the element selected in the Java editor or in a Java view.

	
Javadoc view	<p>There is a Javadoc view (Window > Show View > Other > Java > Javadoc) which shows the Javadoc of the element selected in the Java editor or in a Java view. The Javadoc view uses the SWT Browser widget to display HTML on platforms which support it.</p> 
Type Hierarchy view and method implementations / definitions	<p>To find out which types in a hierarchy override a method, use the 'Show Members in Hierarchy' feature.</p> <ul style="list-style-type: none"> • Select the method to look at and press F4 (Navigate > Open Type Hierarchy). This opens the type hierarchy view on the method's declaring type. • With the method selected in the Hierarchy view, press the 'Lock View and Show Members in Hierarchy' tool bar button. • The hierarchy view now shows only types that implement or define the 'locked' method. You can for example see that 'isEmpty()' is defined in 'List' and implemented in 'ArrayList' and 'Vector' but not in 'AbstractList'.

	
<p>Type hierarchy view supports grouping by defining type</p>	<p>The type hierarchy method view lets you sort the selected type's methods by its defining types. For example, for AbstractList you can see that it contains methods that were defined in Object, Collection and List:</p> 
<p>Tricks in the type hierarchy</p>	<ul style="list-style-type: none"> • Focus the type hierarchy on a new type by pressing F4 (Navigate > Open Type Hierarchy) on an element or a selected name. • You can open the Hierarchy view not only on types but also on packages, source folders, JAR archives and Java projects. • You can Drag & Drop an element onto the Hierarchy view to focus it on that element. • You can change the orientation (from the default vertical to horizontal) of the Hierarchy view from the view's toolbar menu.
<p>Structural compare of Java source</p>	<p>A structural comparison of Java source ignores the textual order of Java elements like methods and fields and shows more clearly which elements were changed, added, or removed.</p> <p>For initiating a structural comparison of Java files you have two options:</p> <ul style="list-style-type: none"> • Select two Java compilation units and choose Compare With > Each Other from the view's context menu. If the files have differences, they are opened into a Compare Editor. The top pane shows the differing Java elements; double clicking on one of them

shows the source of the element in the bottom pane.

- In any context where a file comparison is involved (e.g. a CVS Synchronization) a double click on a Java file not only shows the content of the file in a text compare viewer, but it also performs a structural compare and opens a new pane showing the results.



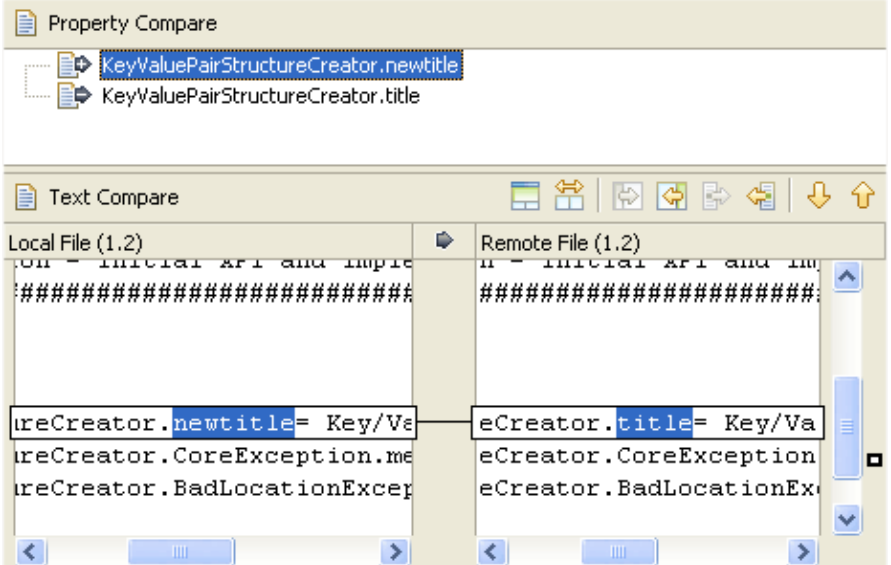
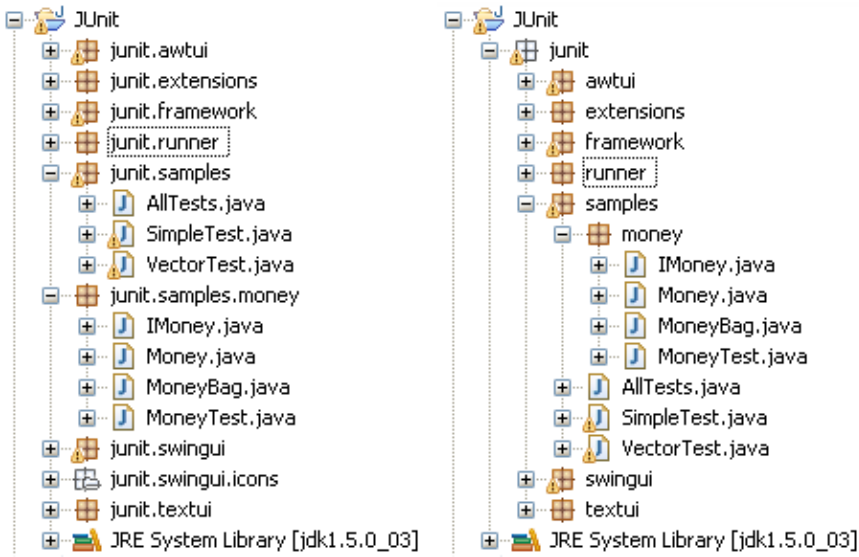
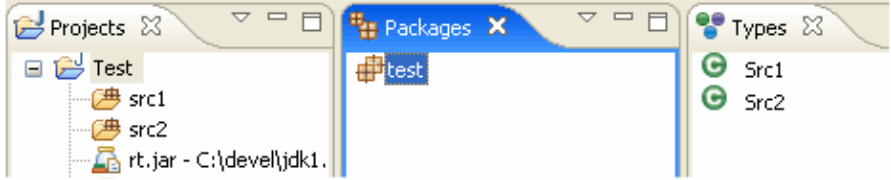
You can even ignore comments and formatting changes when performing the structural compare: turn on the **Ignore Whitespace** option via the Compare Editor's toolbar button, or the CVS Synchronization View's drop down menu.

Structural compare of property files

A structural comparison of Java property files (extension: .properties) ignores the textual order of properties and shows which properties were changed, added, or removed.

For initiating a structural comparison of property files you have two options:


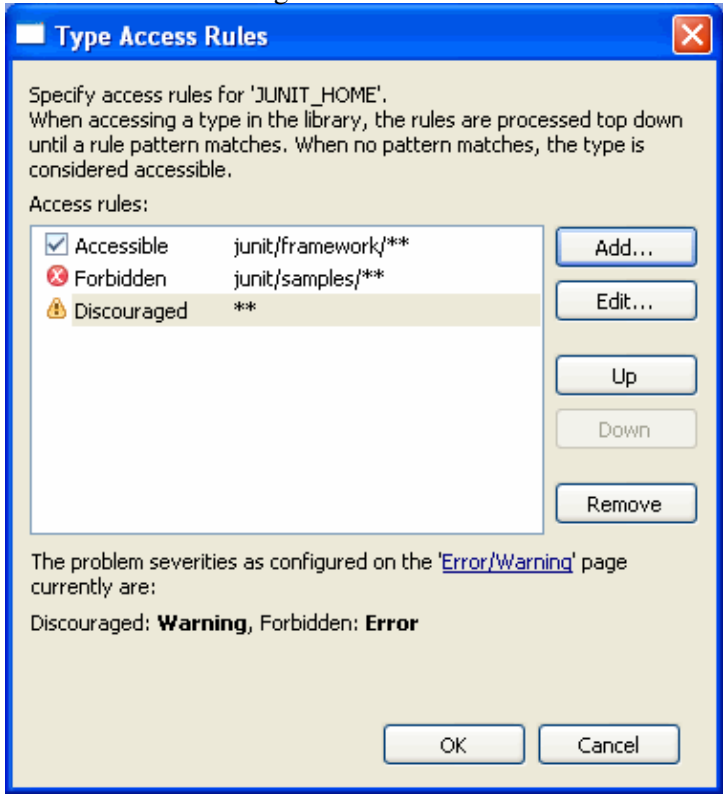



- Select two files in the Package Explorer or Navigator and choose **Compare With > Each Other** from the view's context menu.
- In any context where a file comparison is involved (e.g. a CVS Synchronization) a double click on a property file not only shows the content of the file in a text compare viewer, but it also performs a structural compare and opens a new pane showing the results.


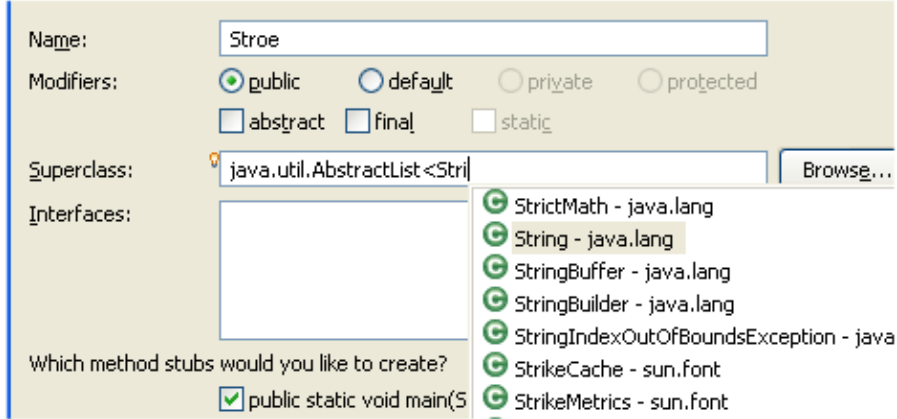
	
<p>Hierarchical vs. flat layout of packages</p>	<p>An option on the Java Packages view (and Package Explorer view) allows you to change the way packages are displayed. Hierarchical displays packages in a tree, with sub-packages below packages; Flat displays them in the standard arrangement, as a flat list with all packages and sub-packages as siblings.</p> 
<p>Logical packages</p>	<p>The Java Packages view (Java Browsing perspective) coalesces packages of the same name across source folders within a project. This shows the Packages view containing a logical package.</p> 

<div>Compress package names</div>	<div><p>If your package names are very long you can configure a compressed name that appears in the viewers. Configuration of the compression pattern is done in > Window > Preferences > Java > Appearance</p><div><div><input checked="" type="checkbox"/> Compress all package name segments, except the final segment</div><div>Compression pattern (e.g. given package name 'org.eclipse.jdt' pattern '.' will compress it to '..jdt', '0' to 'jdt', '1~.' to 'o~.e~.jdt'):</div><div><div>1.</div></div></div><p>Using this example, packages are rendered the following way:</p><div><div><div><div><div></div><div></div></div><div>o.e.s.i.core</div></div><div><div><div></div><div></div></div><div>o.e.s.i.c.text</div></div><div><div><div></div><div></div></div><div>o.e.s.i.ui</div></div><div><div><div></div><div></div></div><div>o.e.s.i.u.text</div></div><div><div><div></div><div></div></div><div>o.e.s.i.u.util</div></div><div><div><div></div><div></div></div><div>o.e.s.ui</div></div><div><div><div></div><div></div></div><div>o.e.s.u.text</div></div><div><div><div></div><div></div></div><div>o.e.s.i.ui</div></div><div><div><div></div><div></div></div><div>o.e.s.i.u.b.views</div></div><div><div><div></div><div></div></div><div>o.e.s.i.u.text</div></div></div></div></div>
<div><div>Manipulating the Java build path directly in the package explorer</div><div>3.1</div></div>	<div><p>Instead of manipulating the Java Build path on Project > Properties > Java Build Path, use the actions in the package explorer's context menu. You can for example add new source folders, archives and libraries to the build path or in– and exclude file and folders from a source folder.</p><div><div><div><div>Build Path</div><div>SourceAlt+Shift+S</div><div>RefactorAlt+Shift+T</div><div>Import...</div><div>Export...</div><div>RefreshF5</div></div><div><div><div>Link Additional Source to Project</div><div>Use as Source Folder</div><div>Add External Archives</div><div>Add Libraries</div><div>Configure Build Path...</div></div></div></div></div></div>

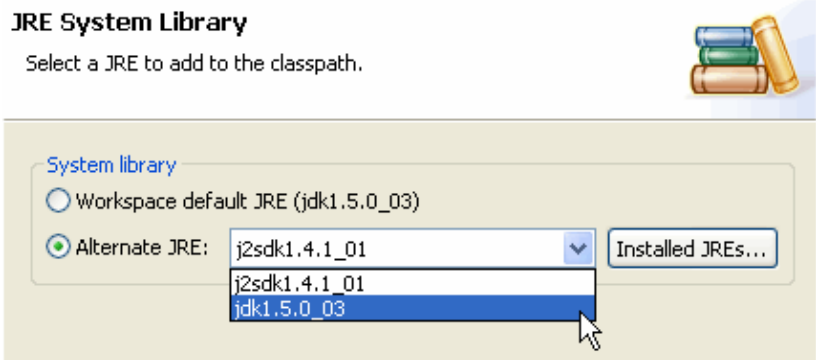
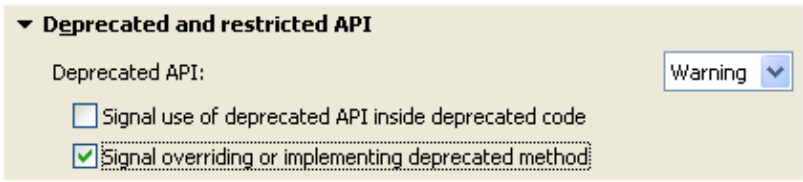
Miscellaneous

Project specific preferences <div data-bbox="212 1539 331 1564" style="background-color: #4a4a8a; color: white; padding: 2px;">3.1</div>	<p>All code style and compiler options can now be defined per project. Open the project property pages with Project > Properties on a project or use the link on the workspace preferences (e.g. Window > Preferences > Java > Code Style) to open a project property page and enable project specific settings.</p> <div data-bbox="451 1638 1235 1831"> <div> Formatter ↶ ↷ </div> <div> <input checked="" type="checkbox"/> Enable project specific settings </div> <div> <div>MyProfile</div> <div>Edit...</div> <div>Rename...</div> <div>Remove</div> </div> </div> <p>The project specific settings are stored in a configuration file inside the project (in the '.settings' folder). When you share a project in a team, team</p>
--	---

	members will also get these project specific settings.
Access Rules 	<p>Access rules give you the possibility to enforce API rules for types from referenced libraries. On the Java build path page (Project > Properties > Java Build Path) edit the 'Access Rules' node available on referenced projects, archives, class folders and libraries.</p> <p>Packages or types in these references can be classified as:</p> <ul style="list-style-type: none"> • Accessible • Discouraged • Forbidden <p>According to the settings on Window > Preferences > Java > Compiler > Errors/Warnings, the compiler will mark discouraged and forbidden references with warning or errors.</p> 
JUnit	<p>Select a JUnit test method in a view and choose Run > JUnit Test from the context menu or Run > Run As > JUnit Test from the main menu. This creates a launch configuration to run the selected test.</p>
Hide JUnit view until errors or failures occur	<p>You can make the JUnit view open only when there are errors or failures. That way, you can have the view set as a fast view and never look at it when there are no failing tests. While there are no problems in your tests you will see this icon  (the number of small green squares will grow, indicating progress) while running them and this icon  after they are finished. If, however, errors or failures occur, the icon will change to  (or</p>

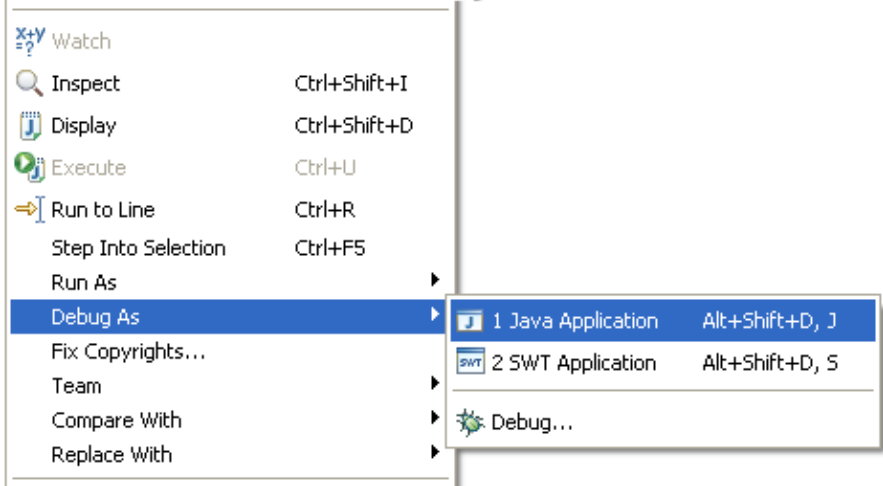
	<p> if tests are finished) and the JUnit view will appear. This behavior can be configured via the Window > Preferences > Java > JUnit preference page.</p>
<p>Content assist in dialog fields</p>	<p>Content Assist (Ctrl+Space) is now also available in input fields of various Java dialogs. Look for small light bulb icon beside the field when it has focus.</p>  <p>Content Assist is e.g. implemented in the New Java Class, New Java Interface, and New JUnit Test wizards, as well as in the refactoring dialogs for Change Method Signature and moving static members.</p> <p>The Extract Local Variable, Convert Local Variable to Field, and Introduce Parameter refactorings offer content assist proposals for the new element name.</p>
<p>Define prefixes or suffixes for fields, parameters and local variables</p>	<p>In addition to configuring the prefix or suffix for fields, you can also specify the prefix or suffix for static fields, parameters, and local variables. These settings on the Window > Preferences > Java > Code Style preference page are used in content assist, quick fix, and refactoring whenever a variable name is computed.</p>

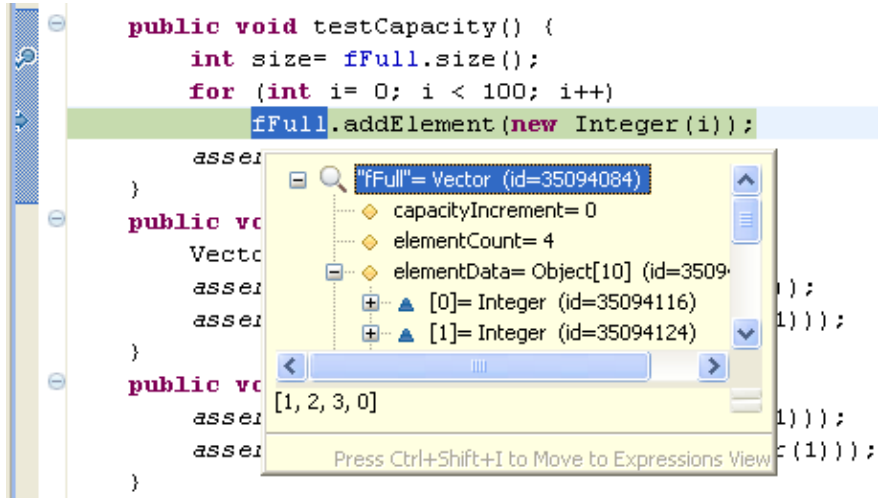
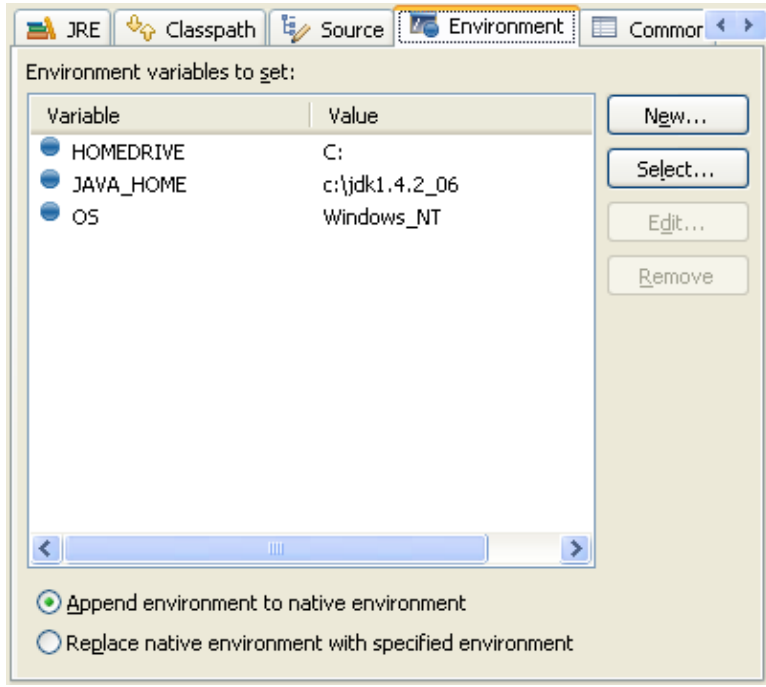
	<div><div>Code Style</div><div><div>Configure Project Specific Settings...</div><div>Conventions for variable names:</div><table><thead><tr><th>Variable type</th><th>Prefix list</th><th>Suffix list</th></tr></thead><tbody><tr><td>Fields</td><td>f</td><td></td></tr><tr><td>Static Fields</td><td>fg</td><td></td></tr><tr><td>Parameters</td><td></td><td></td></tr><tr><td>Local Variables</td><td></td><td></td></tr></tbody></table><div><div><input type="checkbox"/> Qualify all generated field accesses with 'this.'</div><div><input checked="" type="checkbox"/> Use 'is' prefix for getters that return boolean</div><div><input type="checkbox"/> Automatically add comments for new methods and types</div><div><input checked="" type="checkbox"/> Add '@Override' annotation for new overriding methods</div></div><div>Exception variable name in catch blocks: <input type="text" value="e"/></div></div></div>	Variable type	Prefix list	Suffix list	Fields	f		Static Fields	fg		Parameters			Local Variables		
Variable type	Prefix list	Suffix list														
Fields	f															
Static Fields	fg															
Parameters																
Local Variables																
Organize Imports works on more than single files	You can invoke Organize Imports on sets of compilation units, packages, source folders or Java projects.															
Format more than one file	Select all Java files to format and choose Source > Format to format them all. The format action is also available on packages, source folders or Java projects.															
Use project specific compiler settings	Each project can use the global compiler settings or you can define project specific settings. Select the project and open the Java compiler page in the project properties (Project > Properties > Java Compiler)															

<p>Use a specific JRE for a project</p>	<p>When creating new projects the JRE that is added by default is the one selected in Window > Preferences > Java > Installed JRE's. To set a project specific JRE, open the project's Java Build path property page (Project > Properties > Java Build Path), then the Libraries page, select 'JRE System Library' and press Edit. In the 'Edit Library' dialog you can select either the default JRE or a project specific JRE to add to new projects.</p> <p>JRE System Library Select a JRE to add to the classpath.</p> 
<p>Propagating deprecation tag</p>	<p>The Java compiler can be configured to diagnose deprecation using options on the Window > Preferences > Java > Compiler > Advanced page.</p>  <p>Using this configuration, the result is:</p> <pre>public interface I { /** @deprecated */ void foo(); } public class X implements I { public void foo() {} } // The method X.foo() overrides a deprecated method from I public class Y extends X { public void foo() {} }</pre> <p>If you're unable to fix a usage of a deprecated construct, we recommend that you tag the enclosing method, field or type as deprecated. This way, you acknowledge that you did override a deprecated construct, and the deprecation flag is propagated to further dependents.</p>

	<pre> public class X implements I { /** @deprecated */ public void foo() {} } public class Y extends X { public void foo() {} } </pre> <p>The method Y.foo() overrides a deprecated method from X</p>
Recovering from abnormal inconsistencies	<p>In the rare event of a dysfunction, JDT could reveal some inconsistencies such as:</p> <ul style="list-style-type: none"> • missing results in a Java Search or Open Type • invalid items in package explorer <p>To make it consistent again, the following actions need to be performed in this exact order:</p> <ol style="list-style-type: none"> 1. Close all projects using navigator Close Project menu action 2. Exit and restart Eclipse 3. Open all projects using navigator Open Project menu action 4. Manually trigger a clean build of entire workspace (Project > Clean...)

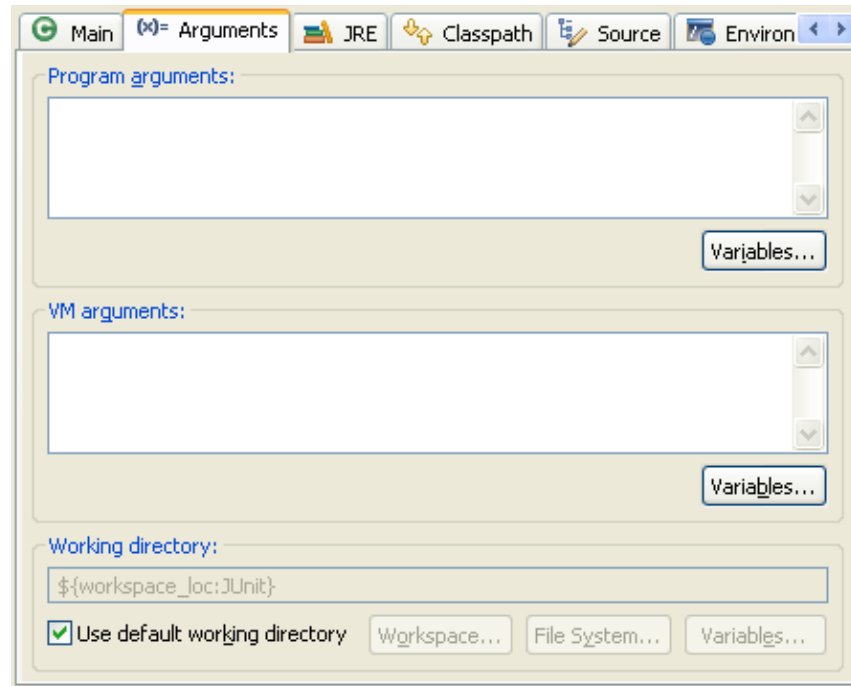
Debugging

Launching from the Context Menu	<p>You can run and debug Java applications from the context menu. You can launch a source file, package, method, type, etc. by choosing Run As (or Debug As) > Java Application from the context menu in a view or editor. Alternatively, you can use the Java application launch shortcut key binding (Alt+Shift+D, J). The top level Run As (or Debug As) actions are also sensitive to the current selection or active editor.</p> 
--	--

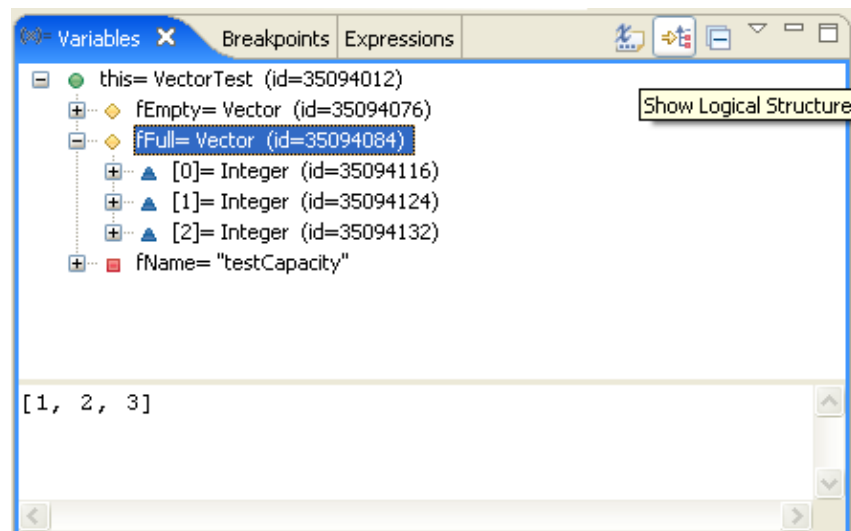
Evaluations in the debugger	<p>Snippet evaluation is available from a number of places in the debugger. Choosing Display or Inspect from the context menu of the editor or Variables view will show the result in a pop-up whose result can be sent to the Display or Expressions views.</p> 
View Management in Non-Debug perspectives	<p>The Debug view automatically manages debug related views based on the view selection (displaying Java views for Java stack frames and C views for C stack frames, for example). By default, this automatic view management only occurs in the Debug perspective, but you can enable it for other perspectives via the View Management preference page available from the Debug view toolbar pulldown.</p>
Environment Variables	<p>You can now specify the environment used to launch Java applications via the Environment tab.</p> 

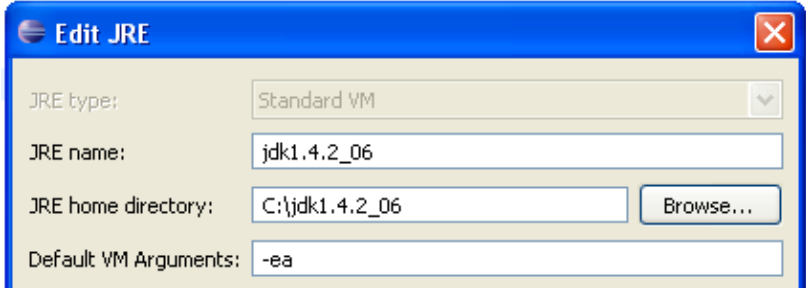
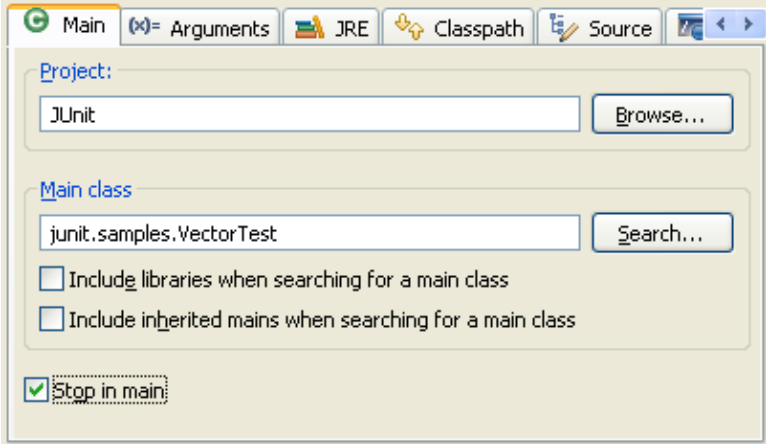
String Substitutions

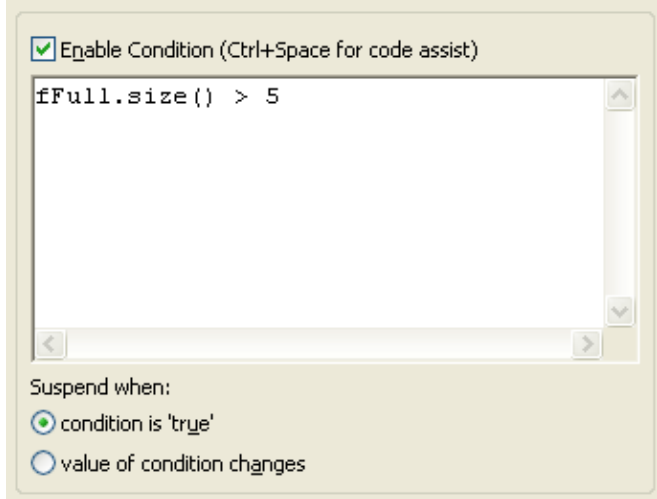
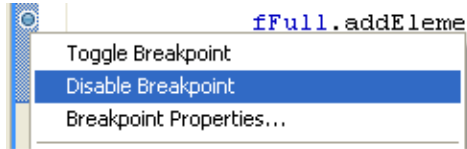
Variables are now supported for many parameters of Java Application launch configurations. Most fields that support variables have a **Variables...** button next to them, such as the program and VM arguments fields. The **Main Type** field supports variables as well; the `${java_type_name}` variable allows you to create a configuration that will run the selected type.

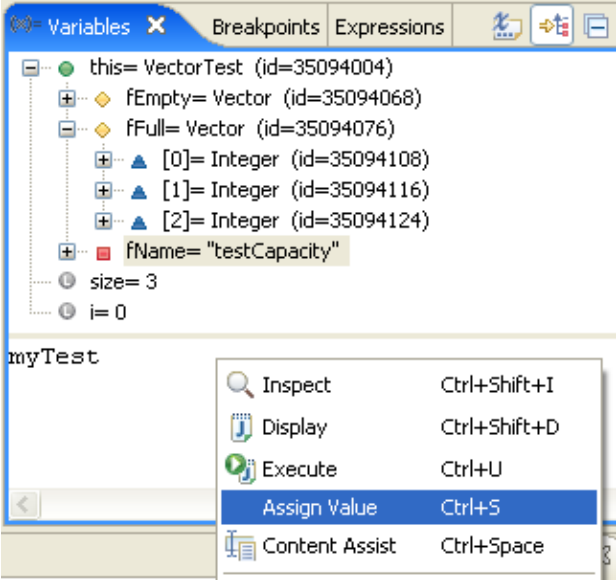
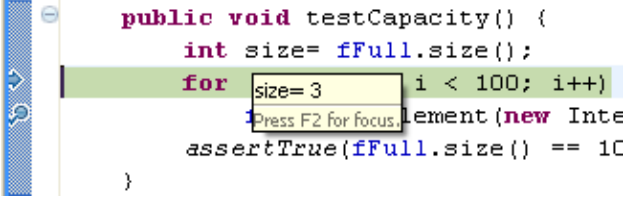
**Logical Structures**

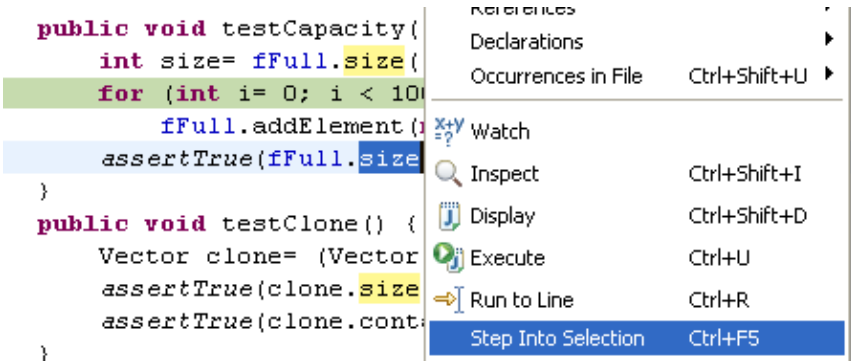
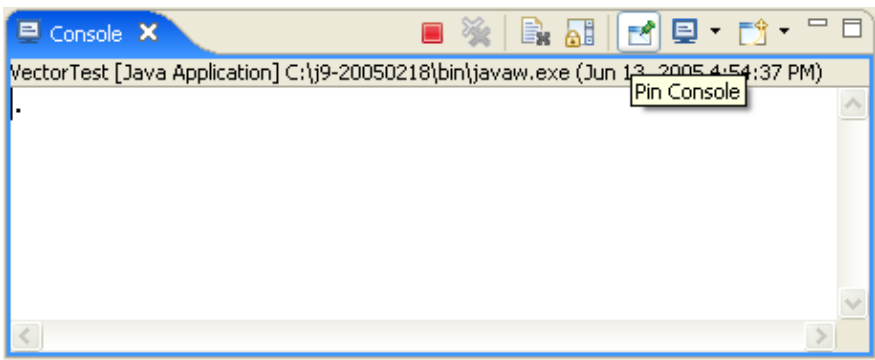
The **Logical Structures** toggle on the **Variables view** toolbar presents alternate structures for common types. JDT provides logical views for Maps, Collections, and SWT Composites. You can define logical structures for other types from the **Logical Structures** preference page.

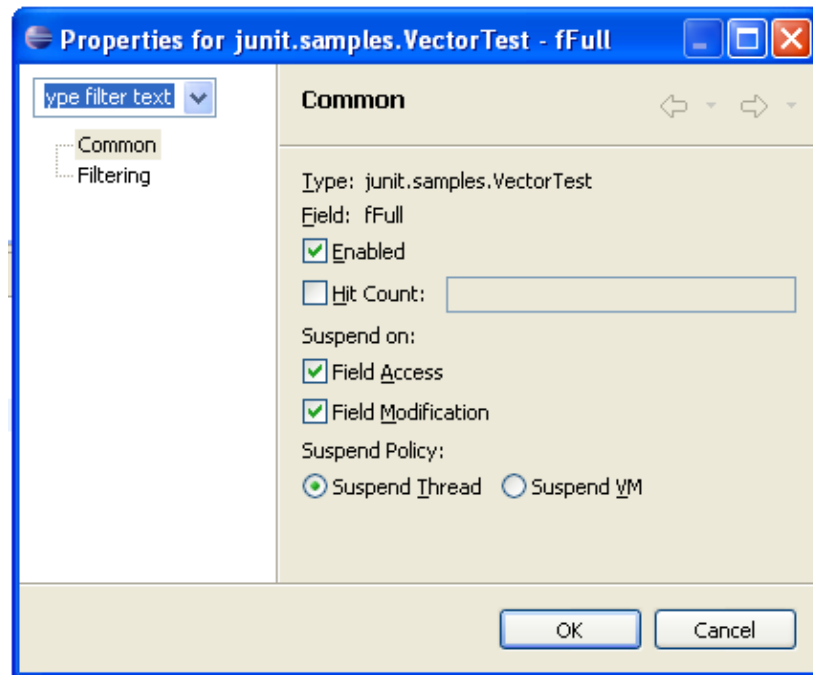


Default VM Arguments	<p>If you specify the same arguments to a certain VM frequently, you can configure Default VM Arguments in the Installed JREs preference page. This is more convenient than specifying them for each launch configuration.</p> 
Stop in Main	<p>You can use Stop in main in a Java Application launch configuration to cause your program to stop at the first executable line of the main method when you run it under debug mode.</p> 
Conditional breakpoints	<p>You can use conditional breakpoints in Breakpoint Properties... to control when a breakpoint actually halts execution. You can specify whether you want the breakpoint to suspend execution only when the condition is true, or when the condition value changes.</p>

	
<p>Disabling breakpoints</p>	<p>If you find yourself frequently adding and removing a breakpoint in the same place, consider disabling the breakpoint when you don't need it and enabling it when needed again. This can be done using Disable Breakpoint in the breakpoint context menu or by unchecking the breakpoint in the Breakpoints view.</p> <p>You can also temporarily disable all breakpoints using the Skip All Breakpoints action in the Breakpoints view toolbar. This will tell the debugger to skip all breakpoints while maintaining their current enabled state.</p> 
<p>Changing variable values</p>	<p>When a thread is suspended in the debugger, you can change the values of Java primitives and Strings in the Variables view. From the variable's context menu, choose Change Variable Value. You can also change the value by typing a new value into the Details pane and using the Assign Value action in the context menu (CTRL-S key binding).</p>

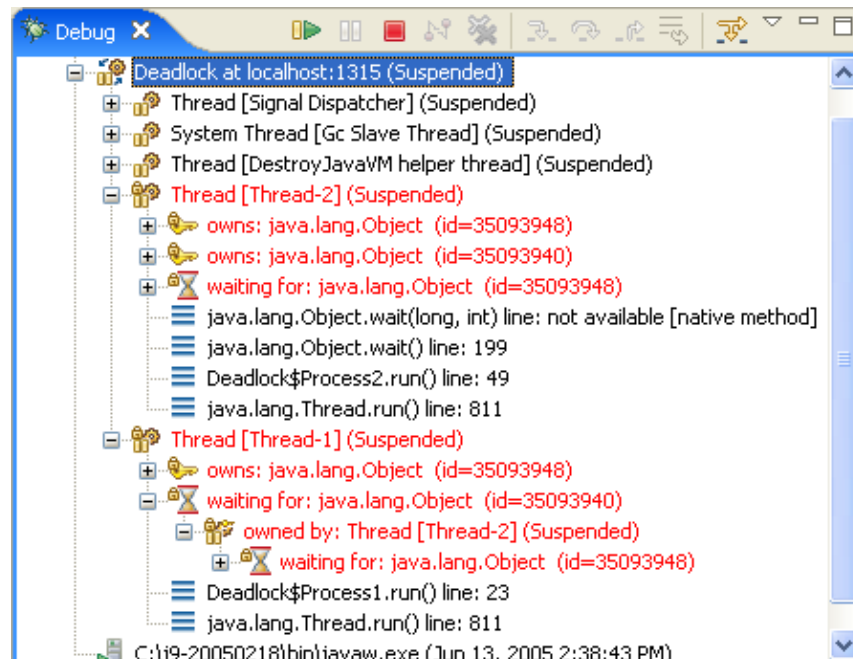
	
Variable values in hover help	<p>When a thread is suspended and the cursor is placed over a variable in the Java editor, the value of that variable is displayed as hover help.</p> 
Drop to Frame	<p>When stepping through your code, you might occasionally step too far, or step over a line you meant to step into. Rather than restarting your debug session, you can use the Drop to Frame action to quickly go back to the beginning of a method. Select the stack frame corresponding to the Java method you wish to restart, and select Drop to Frame from Debug view toolbar or the stack frame's context menu. The current instruction pointer will be reset to the first executable statement in the method. This works for non-top stack frames as well.</p> <p>Note that Drop to frame is only available when debugging with a 1.4 or higher VM, or the J9 VM. There are some situations where a JVM may be unable to pop the desired frames from the stack. For example, it is generally impossible to drop to the bottom frame of the stack or to any frame below a native method.</p>
Hot code replace	<p>The debugger supports Hot Code Replace when debugging with a 1.4 or higher VM, or the J9 VM. This lets you make changes to code you are currently debugging. Note that some changes such as new or deleted methods, class variables or inner classes cannot be hot swapped, depending on the support provided by a particular VM.</p>

Stepping into selections	<p>The Java debugger allows you to step into a single method within a series of chained or nested method calls. Simply highlight the method you wish to step into and select Step into Selection from the Java editor context menu.</p> <p>This feature works in places other than the currently executing line. Try debugging to a breakpoint and stepping into a method a few lines below the currently instruction pointer.</p>  <pre> public void testCapacity() { int size= fFull.size() for (int i= 0; i < 100; i++) { fFull.addElement(i); assertTrue(fFull.size() > 0); } } public void testClone() { Vector clone= (Vector) fFull.clone(); assertTrue(clone.size() > 0); assertTrue(clone.containsAll(fFull)); } </pre>
Controlling your console	<p>Output displayed in the console can be locked to a specific process via the Pin Console action in the Console view toolbar. There's also a Scroll Lock action that stops the console from automatically scrolling as new output is appended.</p> 
Creating watch items	<p>A watch item is an expression in the Expressions view whose value is updated as you debug. You can create watch items from the Java editor by selecting an expression or variable and choosing Watch from its context menu or the top-level Run menu.</p>
Watch points	<p>A watch point is a breakpoint that suspends execution whenever a specified variable is accessed or modified. To set a watchpoint, select a variable in the Outline view and choose Toggle Watchpoint from its context menu. To configure a watchpoint, select the watchpoint in the Breakpoints view and choose Properties... from its context menu. The most important properties for this type of breakpoint are the Access and Modification checkboxes which control when the breakpoint can suspend execution.</p>



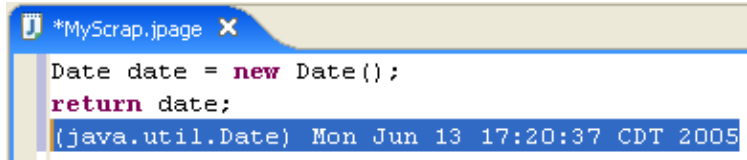
Threads and Monitors

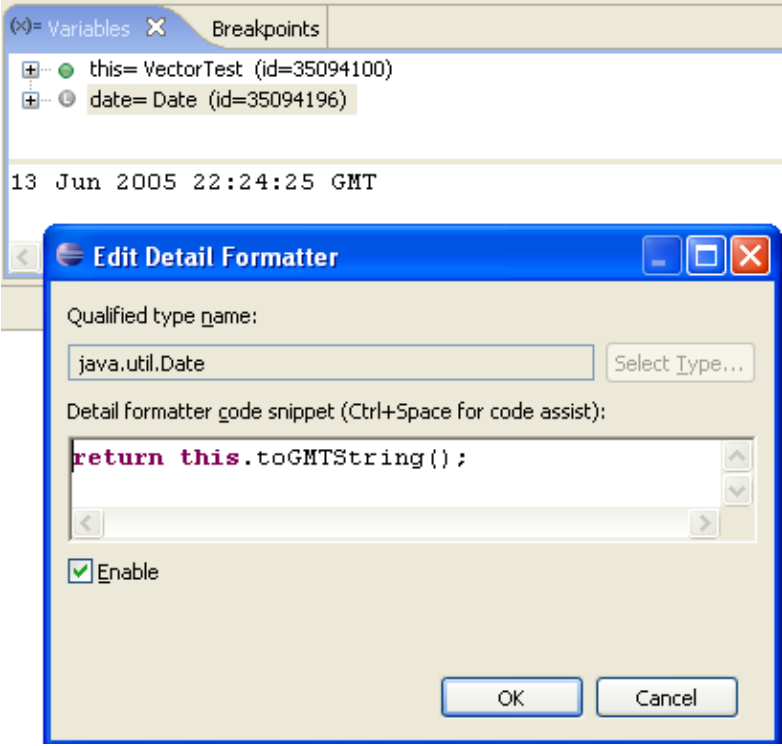
The Java debugger optionally displays monitor information in the **Debug** view. Use the **Show Monitors** action in the Debug view drop down menu to show which threads are holding locks and which are waiting to acquire locks. Threads involved in a deadlock are rendered in red.

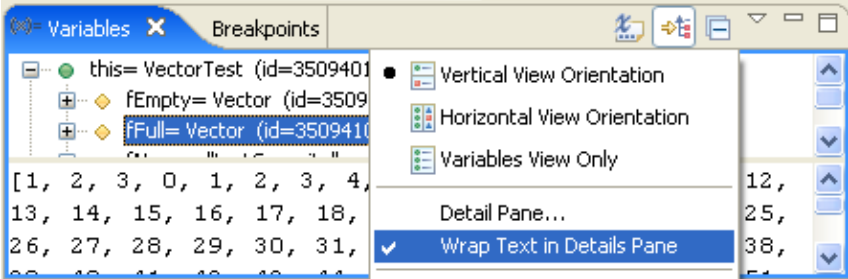
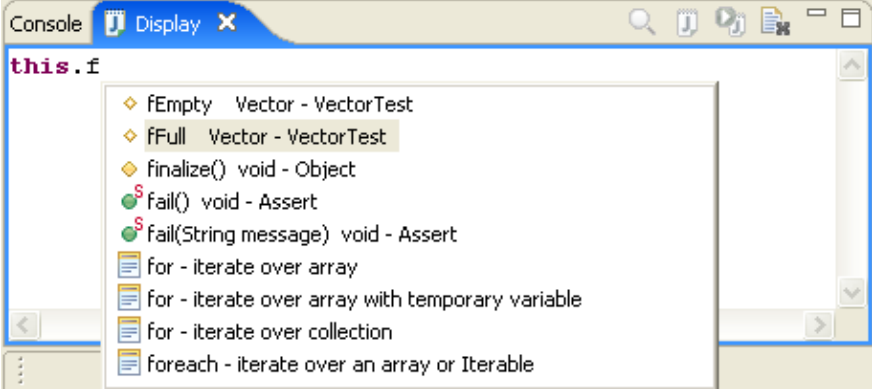
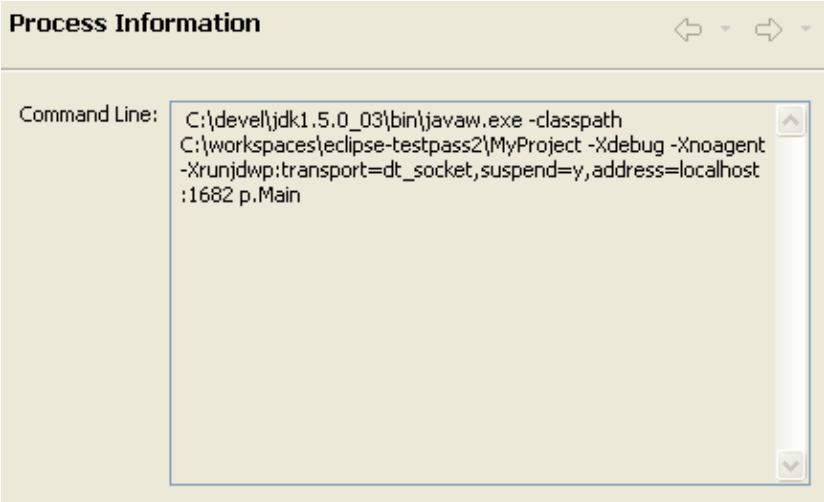


Step filters

Step filters prevent the debugger from suspending in specified classes and packages when stepping into code. Step filters are established in **Window > Preferences > Java > Debug > Step Filtering**. When the **Use Step Filters** toggle (on the debug toolbar and menu) is on, step filters are

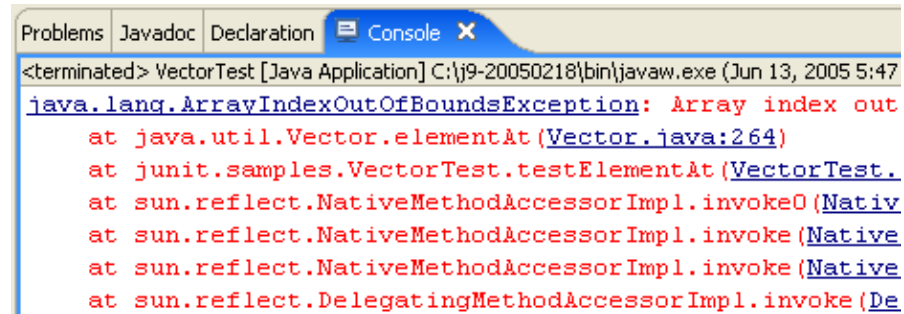
	<p>applied to all step actions. In the Debug view, the selected stack frame's package or declaring type can be quickly added to the list of filters by selecting Filter Type or Filter Package from the stack frame's context menu.</p>
Using the scrapbook	<p>If you want to experiment with API or test out a new algorithm, it's frequently easier to use a Java scrapbook page than create a new class. A scrapbook page is a container for random snippets of code that you can execute at any time without a context. To create a scrapbook page, create a new file with a <i>.jpage</i> extension (or use the <i>New</i> wizard – Java > Java Run/Debug > Scrapbook Page). Enter whatever code you wish to execute, then select it. There are three ways to execute your code:</p> <ul style="list-style-type: none"> • Execute the selected code and place the returned result in the Expressions view • Execute the selected code and place the String result right in the scrapbook page  <ul style="list-style-type: none"> • Execute the selected code (and ignore any returned result) <p>These actions are in the workbench toolbar and also in the scrapbook page's context menu.</p>
Editing launch configurations	<p>Holding down the Ctrl key and making a selection from the Run or Debug drop-down menu opens the associated launch configuration for editing. The launch configuration can also be opened from the context menu associated with any item in the Debug view.</p>
Favorite launch configurations	<p>Launch configurations appear in the Run/Debug drop-down menus in most recently launched order. However it is possible to force a launch configuration to always appear at the top of the drop-downs by making the configuration a 'favorite'. Use the Organize Favorites... action from the appropriate drop down menu to configure your favorite launch configurations.</p>
Detail formatters	<p>In the Variables & Expressions views, the detail pane shows an expanded representation of the currently selected variable. By default, this expanded representation is the result of calling toString() on the selected object, but you can create a custom detail formatter that will be used instead by choosing New Detail Formatter from the variable's context menu. This detail formatter will be used for all objects of the same type. You can view and edit all detail formatters in the Window > Preferences > Java > Debug > Detail Formatters preference page.</p>

	
<p>Running code with compile errors</p>	<p>You can run and debug code that did not compile cleanly. The only difference between running code with and without compile errors is that if a line of code with a compile error is executed, one of two things will happen:</p> <ul style="list-style-type: none"> • If the 'Suspend execution on compilation errors' preference on the Window > Preferences > Java > Debug preference page is set and you are debugging, the debug session will suspend as if a breakpoint had been hit. Note that if your VM supports Hot Code Replace, you could then fix the compilation error and resume debugging • Otherwise, execution will terminate with a 'unresolved compilation' error <p>It is important to emphasize that as long as your execution path avoids lines of code with compile errors, you can run and debug just as you normally do.</p>
<p>Word wrap in Variables view</p>	<p>The details area of the debugger's Variables and Expressions views supports word wrap, available from the view drop-down menu.</p>

	
Code assist in the debugger	<p>Code assist is available in many contexts beyond writing code in the Java editor:</p> <ul style="list-style-type: none"> • When entering a breakpoint condition • In the Details pane of the Variables & Expressions view • When entering a Detail Formatter code snippet • When entering a Logical Structure code snippet • When entering code in a Scrapbook page • In the Display view 
Command line details	<p>You can always see the exact command line used to launch a program in run or debug mode by selecting Properties from the context menu of a process or debug target, even if the launch has terminated.</p> 

**Stack trace
hyperlinks**

Java stack traces in the console appear with hyperlinks. When you place the mouse over a line in a stack trace the pointer changes to the hand. Pressing the mouse button opens the associated Java source file and positions the cursor at the corresponding line. Pressing the mouse button on the exception name at the top of the stack trace will create an exception breakpoint.



```
<terminated> VectorTest [Java Application] C:\j9-20050218\bin\javaw.exe (Jun 13, 2005 5:47)
java.lang.ArrayIndexOutOfBoundsException: Array index out
    at java.util.Vector.elementAt (Vector.java:264)
    at junit.samples.VectorTest.testElementAt (VectorTest.
    at sun.reflect.NativeMethodAccessorImpl.invoke0 (Nativ
    at sun.reflect.NativeMethodAccessorImpl.invoke (Native
    at sun.reflect.NativeMethodAccessorImpl.invoke (Native
    at sun.reflect.DelegatingMethodAccessorImpl.invoke (De
```

What's New in 3.1

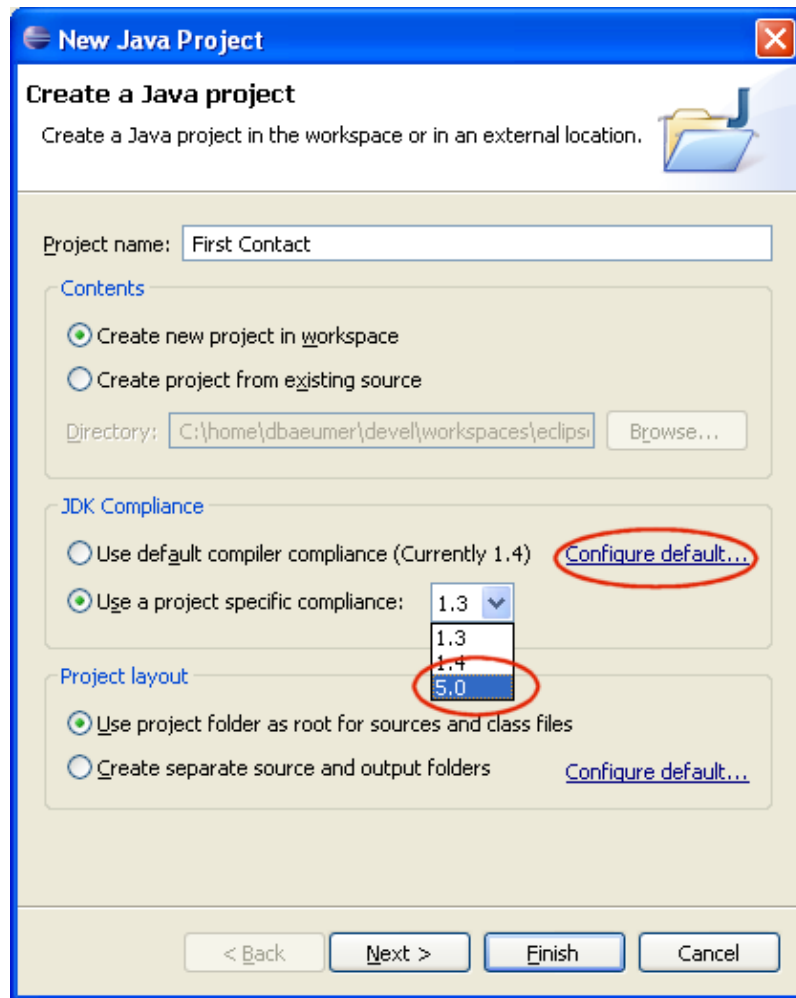
This document contains descriptions of some of the more interesting or significant changes made to the Java development tools for the 3.1 release of Eclipse since 3.0. It is broken into several sections:

- [J2SE 5.0](#)
- [Java Debugger](#)
- [Java Compiler](#)
- [Java Editor](#)
- [General Java Tools](#)

J2SE 5.0

J2SE 5.0 Eclipse 3.1 includes full support for the new features of J2SE 5.0 (aka "Tiger"). One of the most important consequences of this support is that you may not notice it at all — everything that you expect to work with J2SE 1.4, including editing, code assist, compiling, debugging, quick fixes, refactorings, source actions, searching, etc. will work seamlessly with J2SE 5.0's new types and syntax.

In order to develop code compliant with J2SE 5.0, you will need a 5.0 Java Runtime Environment (JRE). If you start Eclipse for the first time using a 5.0 JRE, then it will use it by default. Otherwise, you will need to use the Installed JREs dialog to register one with Eclipse. You can reach this dialog either via the preference **Java > Installed JREs** or by following the **Configure default...** link on the New Java Project wizard.



**Quick Fix to
update JRE and
compiler compliance
to 5.0**

A new quick fix helps you change the compliance settings when you try to enter 5.0 constructs in a 1.4 project. Note that a 1.5 JRE is required, which can be added in the 'Installed JRE's' preference page.

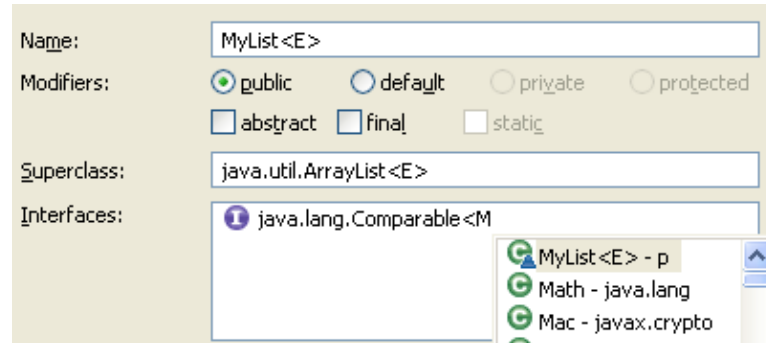
```
public class MyClass {
    List<String> field;
}
```

Change project compliance and JRE to 5.0
Rename in file (Ctrl+2, R direct access)

New Type wizards support generics

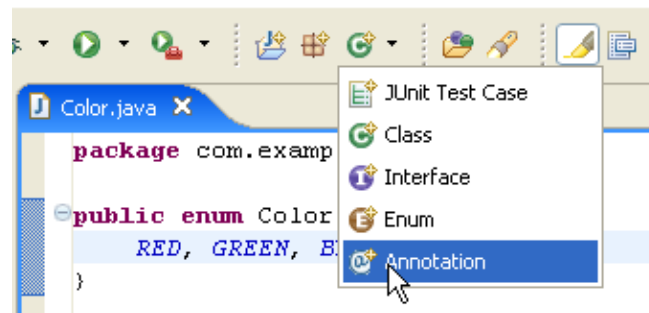
The New Type wizards support J2SE 5.0 generic types in various fields:

- The *Name* field can include type parameter declarations.
- The *Superclass* and the implemented *Interfaces* can include generic type arguments.



Creating Enumerations and Annotations

Enumerations and Annotations can be created with the new Enum or Annotation wizard:



Guessing for type arguments

Code Assist inserts the correct type arguments when completing a type in the Java editor. Type arguments that cannot be disambiguated will be selected, and the **Tab** key will move from one argument to the next.

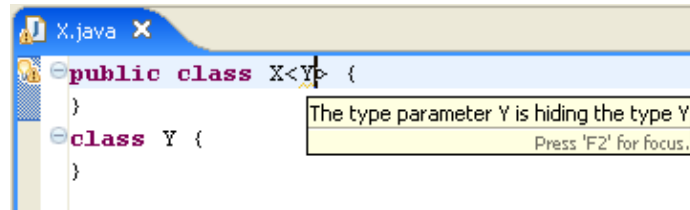
In this example `String` is inserted as the first type argument, while `Number` is proposed for the second:

```
class GenericTypes {
    Map<String, ? extends Number> createMap() {
        return new HashMap<String, Number>();
    }
}
```

To try out this feature, you need to enable **Fill argument names** on the **Java > Editor > Code Assist** preference page.

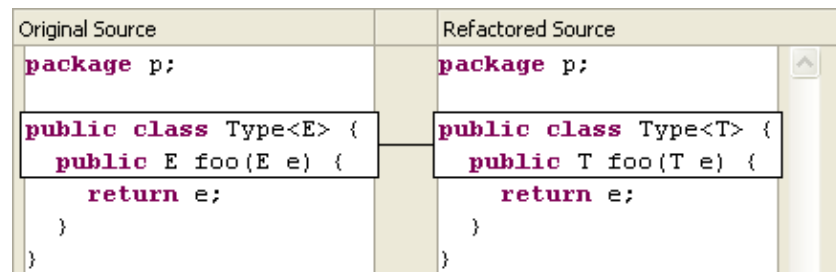
Type parameter declaration hiding another type diagnosis

The Java compiler can optionally flag a type parameter declaration hiding another type.



Rename refactoring

The Rename refactoring has been extended to handle renaming of type parameters.



Infer Generic Type Arguments refactoring

With J2SE 5.0, your code can use generics to enhance readability and static type safety. **Refactor > Infer Generic Type Arguments** is a new refactoring that helps clients of generic class libraries, like the Java Collections Framework, to migrate their code.

```

public class Storage {
    private List fFiles= new ArrayList();

    public File getManagedFile(String name) {
        for (Iterator iter = fFiles.iterator(); iter.hasNext();) {
            File file = (File) iter.next();
            if (file.getName().equals(name))
                return file;
        }
        return null;
    }
    public void manageFile(File file) {
        fFiles.add(file);
    }
}

```

Basic tutorial

The refactoring infers type parameters for generic types, and will remove any unnecessary casts. It works on single compilation units as well as on whole packages and Java projects.

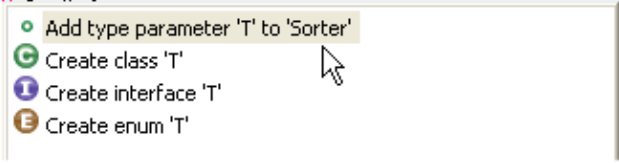
```
public class Storage {
    private List<File> fFiles = new ArrayList<File>();

    public File getManagedFile(String name) {
        for (Iterator<File> iter = fFiles.iterator(); iter.hasNext();) {
            File file = iter.next();
            if (file.getName().equals(name))
                return file;
        }
        return null;
    }
    public void manageFile(File file) {
        fFiles.add(file);
    }
}
```

Quick fixes for Generics

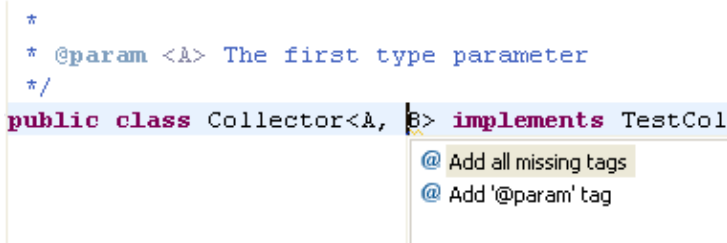
For unresolved Java types, you now also get a proposal to create a new type parameter:

```
public T get() {
}
```



Support for Javadoc tags for type parameters has been added. In J2SE 5.0, you document type parameters using the existing `@param` tag but with the name enclosed in angle brackets.

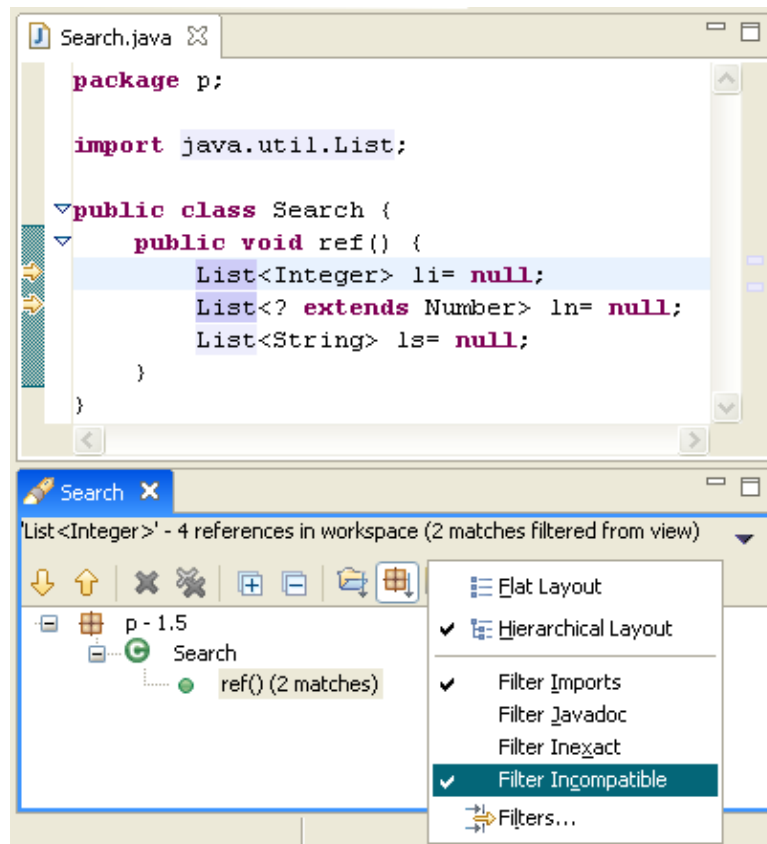
```
*
* @param <A> The first type parameter
*/
public class Collector<A, B> implements TestCol
```



New search result filters for reference search for parameterized types

When searching for references to a parameterized type such as `List<Integer>`, the search result will contain references to all occurrences of `List` as well. The search result view now offers two additional filters to hide matches:

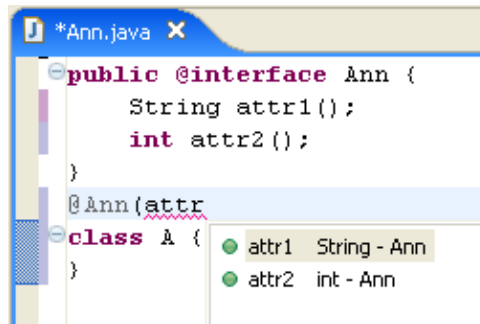
- Filter incompatible matches: this filter hides all results that are not assignment compatible with the search pattern. For example when searching for `List<Integer>` filtering incompatible matches will hide `List<String>`, but not `List<? extends Number>`.
- Filter inexact matches: this filter hides all results that don't exactly match the pattern. For the example above the filter will also hide `List<? extends Number>`.



Completion on annotations

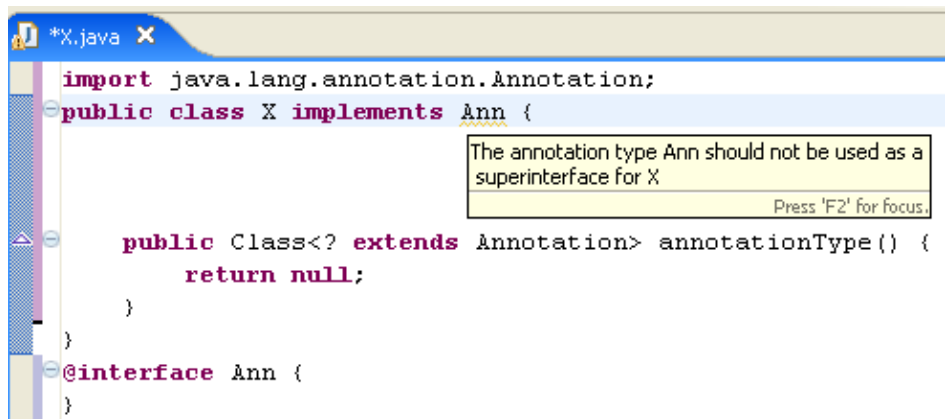
Code completion inside a single member annotation or annotation attribute value is supported.

Basic tutorial



Usage of annotation type as super interface diagnosis

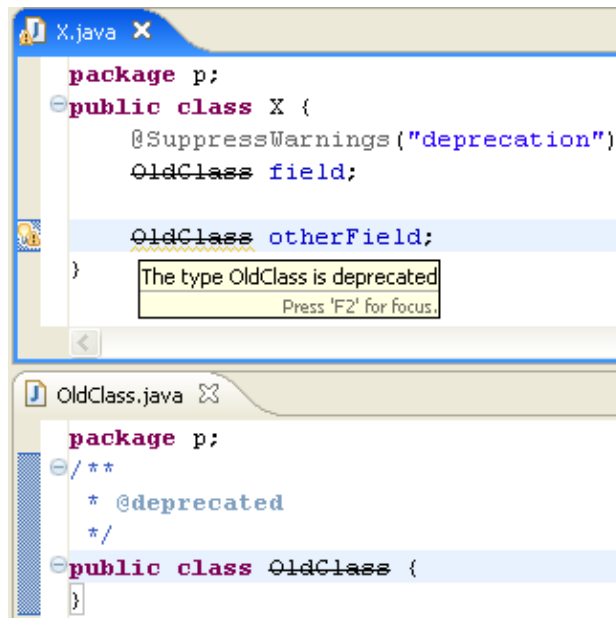
In J2SE 5.0, the Java language allows a class to implement an annotation type. However this should be discouraged. The Java compiler optionally flags such usage.



Support for @SuppressWarnings annotation

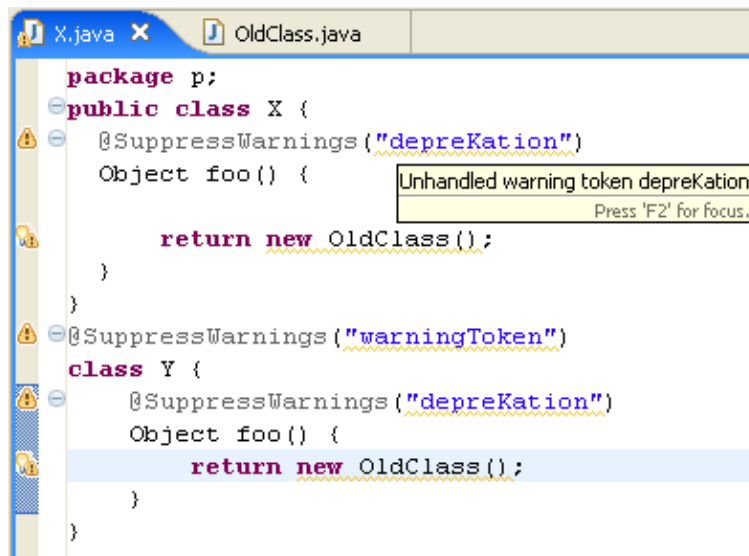
The J2SE 5.0 @SuppressWarnings annotation is supported. Recognized warning token names are: "all", "boxing", "dep-ann", "deprecation", "incomplete-switch", "hiding", "finally", "static-access", "nls", "serial", "synthetic-access", "unqualified-field-access", "unchecked", "unused" and "warningToken". In the example below, the first field is tagged with the @SuppressWarnings("deprecation") annotation and no deprecation warning is reported. The second field is not tagged and a deprecation warning is reported.

Basic tutorial



Note that a compiler option controls whether `@SuppressWarnings` annotations are active or not. See the preference **Java > Compiler > Errors/Warnings > J2SE 5.0 options > Enable '@SuppressWarnings' annotations**

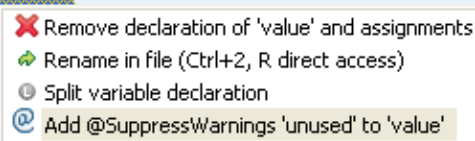
By default, unhandled warning tokens are signaled by a warning. This warning can also be suppressed using the `@SuppressWarnings("warningToken")` annotation.



Quick fix support for @SuppressWarnings

Warnings that can be suppressed using a @SuppressWarnings annotation offer a quick fix to do so. Applying quick fix to the unused local warning below

```
public void foo() {
    String value= null;
}
```



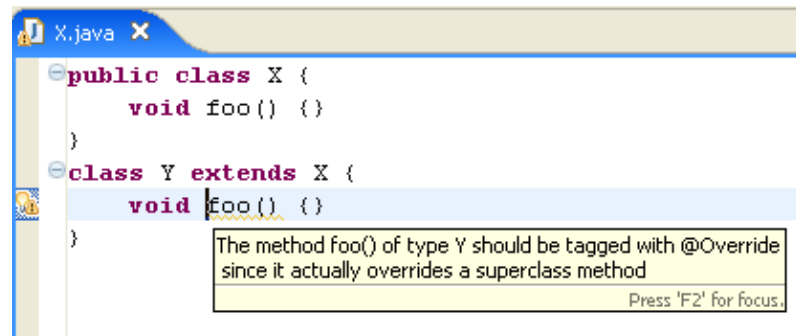
Remove declaration of 'value' and assignments
 Rename in file (Ctrl+2, R direct access)
 Split variable declaration
 Add @SuppressWarnings 'unused' to 'value'

results in:

```
public void foo() {
    @SuppressWarnings("unused") String value= null;
}
```

Missing @Override annotation diagnosis

The Java compiler can optionally flag a method overriding a superclass method, but missing a proper @Override annotation. Missing @Override annotations can be added using Quick Fix.



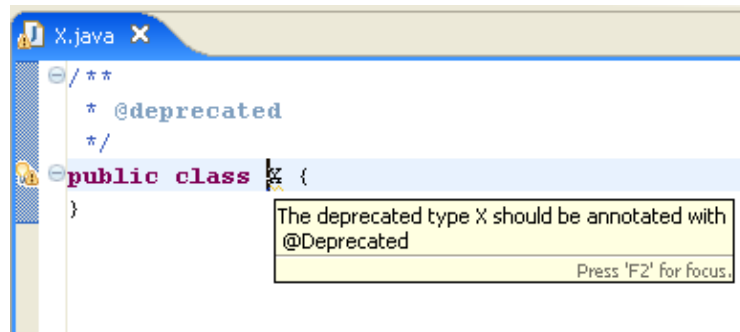
```
X.java
public class X {
    void foo() {}
}
class Y extends X {
    void foo() {}
}
```

The method foo() of type Y should be tagged with @Override since it actually overrides a superclass method
 Press 'F2' for focus.

See the preference **Java > Compiler > Errors/Warnings > J2SE 5.0 options > Missing '@Override' annotation**

Missing @Deprecated annotation diagnosis

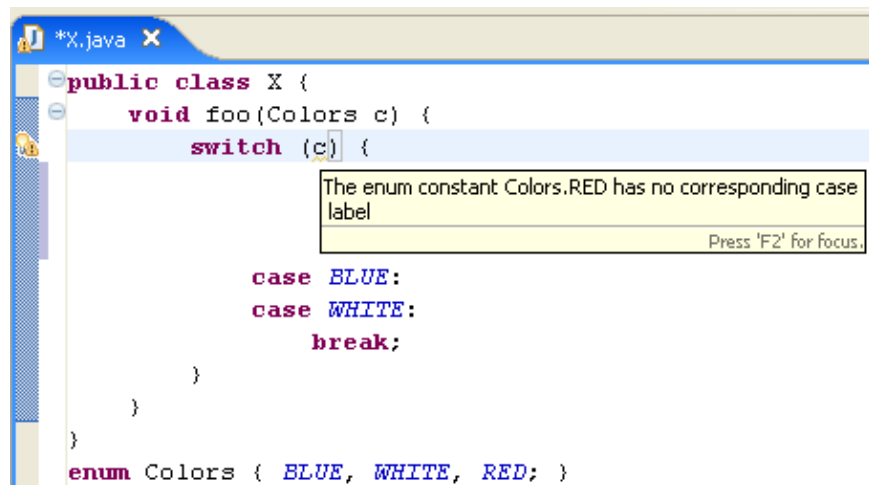
The Java compiler recognizes the @Deprecated annotations, and treats them equivalent to the doc comment `/** @deprecated */`. It can optionally flag deprecated constructs missing a proper @Deprecated annotation (to encourage using annotations instead of doc comment tag).



See preference under **Java > Compiler > Errors/Warnings > J2SE 5.0 options > Missing '@Deprecated' annotation**

Incomplete enum switch statement diagnosis

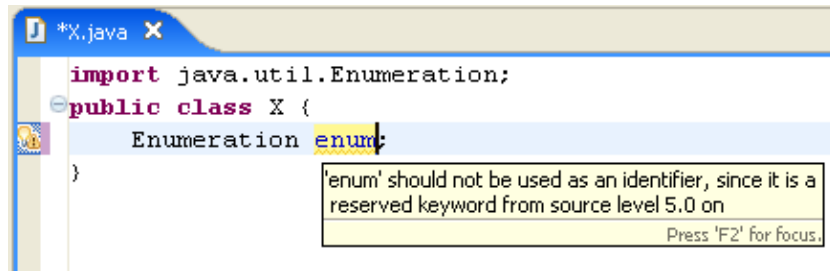
The Java compiler can optionally flag incomplete enum switch statements.



See preference under **Java > Compiler > Errors/Warnings > J2SE 5.0 options > Not all enum constants covered on 'switch'**

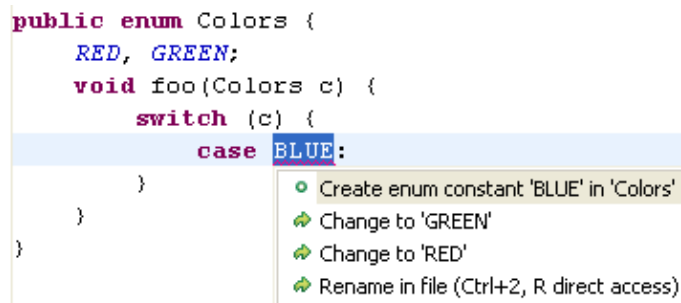
Compiler diagnosis for 'enum' identifier

The Java compiler can find and flag where 'enum' is used as an identifier. While 'enum' is a legal identifier up to source level 1.4, but a reserved keyword in 5.0 source. Enabling this warning helps to anticipate source migration issues. See the preference **Java > Compiler > JDK Compliance > Disallow identifier called 'enum'**.



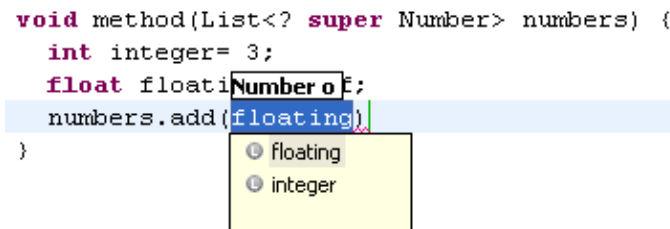
Quick Fix to create enum constants

Quick Fix supports creation of enumeration constants. In the example below the constant BLUE is missing from the enumeration Colors



Autoboxing parameter proposals

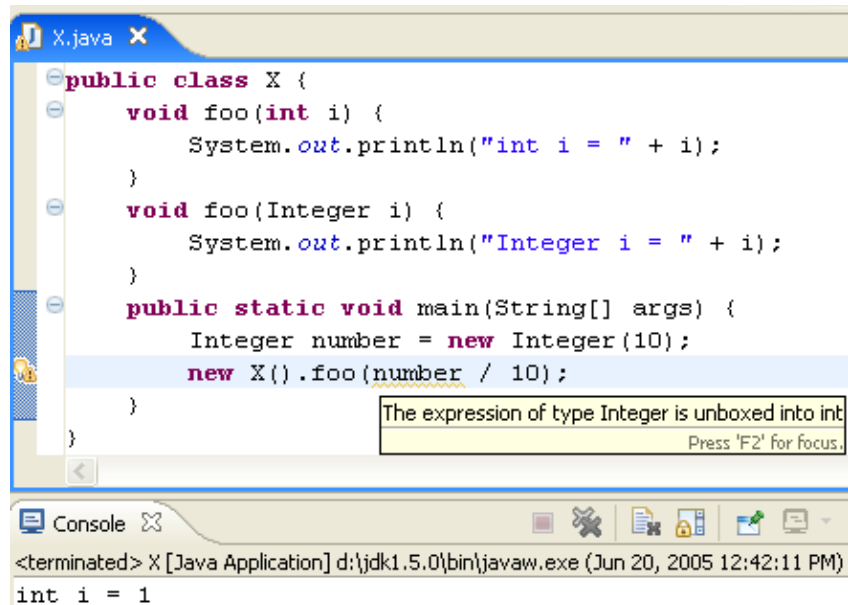
Proposed parameters include auto(un-)boxing proposals:



Note: The **Java > Editor > Code Assist > Fill argument names on completion** preference has to be enabled.

Boxing/unboxing diagnosis

The J2SE 5.0 autoboxing capability is powerful but it can lead to unexpected behavior especially when passing arguments. The compiler introduces an optional diagnosis that indicates when autoboxing or autounboxing is performed. In the following example, one might think that `foo(Integer)` would be called, but since autounboxing is performed, `foo(int)` is called.



See preference under **Java > Compiler > Errors/Warnings > J2SE 5.0 options > Boxing and unboxing conversions**.

Support for J2SE 5.0 in Java editor

The Java editor provides syntax coloring for the new J2SE 5.0 language features. Go to the **Java > Editor > Syntax Coloring** preference page to change the colors or to enable semantic coloring of type variables, annotation elements and auto(un-)boxed expressions:

```
enum TestStyle { CORRECTNESS, PERFORMANCE }

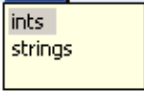
@Target(METHOD)
@interface Test {
    TestStyle style() default TestStyle.CORRECTNESS;
}

@Test(style=TestStyle.PERFORMANCE)
public void randomAccess() {
    List<E> vector= fFull;
    long before= System.currentTimeMillis();
    E one= vector.get(42);
    long after= System.currentTimeMillis();
    Long maxAccessTime= new Long(100);
    assertTrue(after - before < maxAccessTime);
}
```

New for loop template

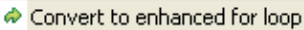
The *foreach* template inserts a new 'for' loop into the code, proposing local Iterable instances you may want to iterate over:

```
void iterate(List<String> strings, Integer... ints) {
    for (Integer integer : ints) {
    }
}
```


Convert to enhanced for loop

A new Quick Assist (**Ctrl+1**) offers to convert old-style for loops over arrays and collections to J2SE 5.0 enhanced for loops:

```
String[] names= {"Calvin", "Hobbes"};
for (int i = 0; i < names.length; i++) {
    String name = names[i];
    System.out.println(name);
}
```

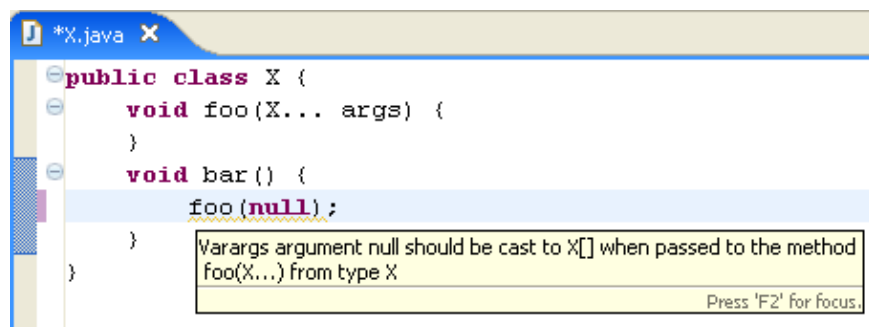


The Quick Fix simplifies the loop to:

```
String[] names= {"Calvin", "Hobbes"};
for (String name : names) {
    System.out.println(name);
}
```

Varargs argument needing a cast

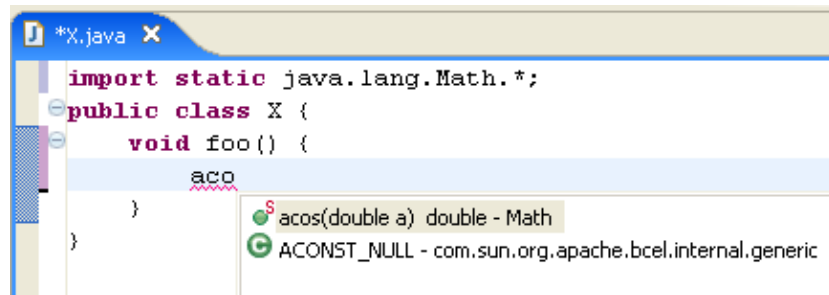
The Java compiler can optionally flag suspicious varargs method invocations. A null last argument is not wrapped as a 1-element array as one might expect; adding an explicit cast makes the intention of the code clear.



The preference setting can be found at **Java > Compiler > Errors/Warnings > J2SE 5.0 Options > Inexact type match for vararg arguments**.

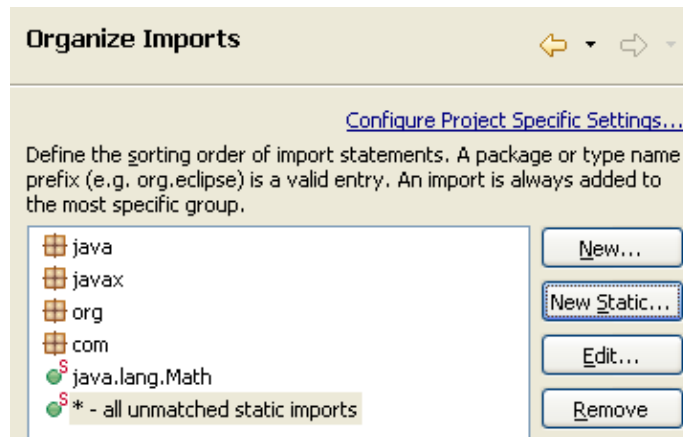
Completion uses static imports

Code completion in the Java editor is able to process static imports when inferring context-sensitive completions.



Static import groups

To organize your static imports, create groups for the static imports and place them where you prefer. You can define an 'others' group to collect up all imports not matched by any other group:



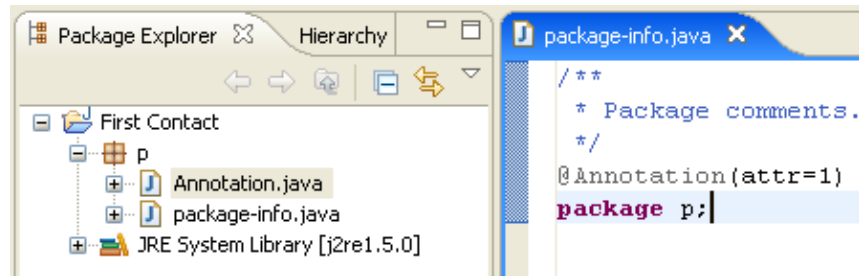
The 'others' group feature is also available for non-static imports.

Support for package-info.java

Support has been added for the special source file package-info.java, which allows annotating and documenting packages. All JDT tools (code assist, code select, search, outline, type hierarchies, etc.) can be used in this special compilation unit.

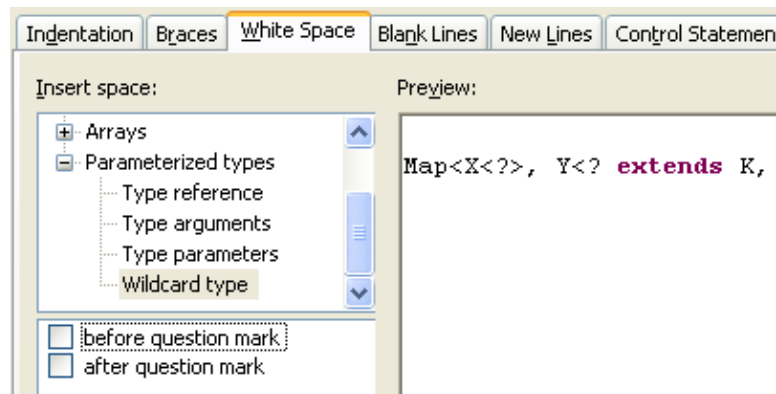
Doc comments inside the package-info.java are processed, and the syntax and references in standard comment tags are verified.

Basic tutorial



Code formatter for J2SE 5.0 constructs

The code formatter supports all the new J2SE 5.0 language constructs. Control over how the formatter handles them are found on the **Java > Code Style > Code Formatter** preference page:

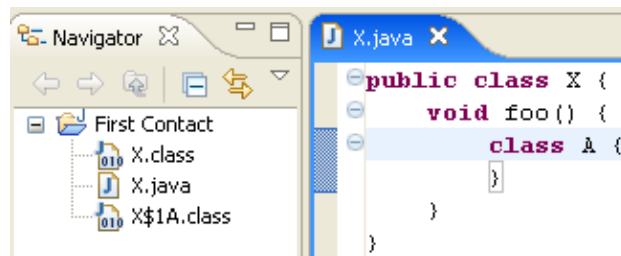


Debugging 5.0 source code

You can run and debug 5.0 source code with a 1.5 JRE. Java debug evaluations support J2SE 5.0 constructs such as generics and enhanced for loops.

Class file naming change for local inner types

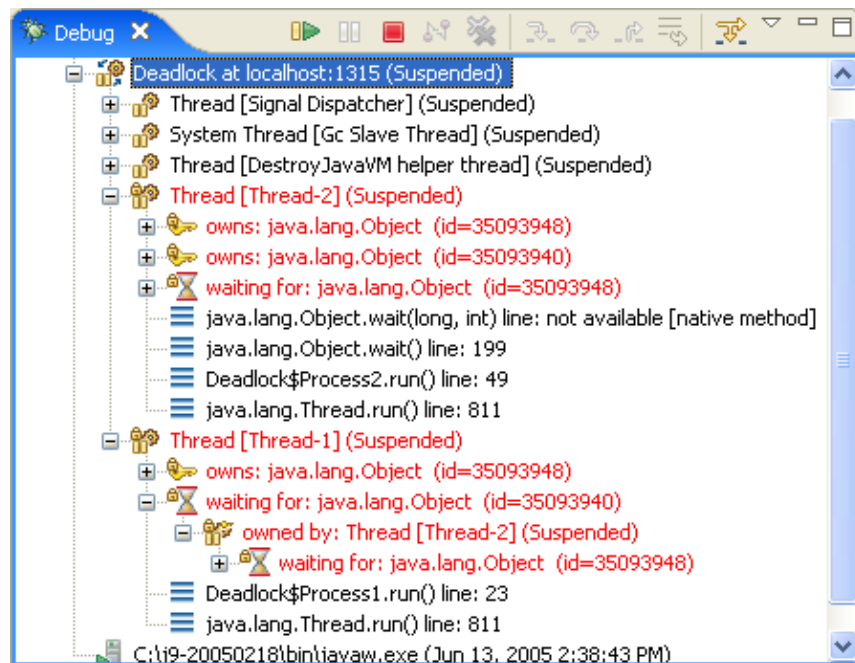
In 5.0 compliance mode, the Java compiler generates class files that follow the naming convention specified in JLS 13.1 (3rd edition) for local inner types. As a consequence, in the below example, instead of generating a file named `X$1A.class`, it will simply be `X$1A.class`.



Java Debugger

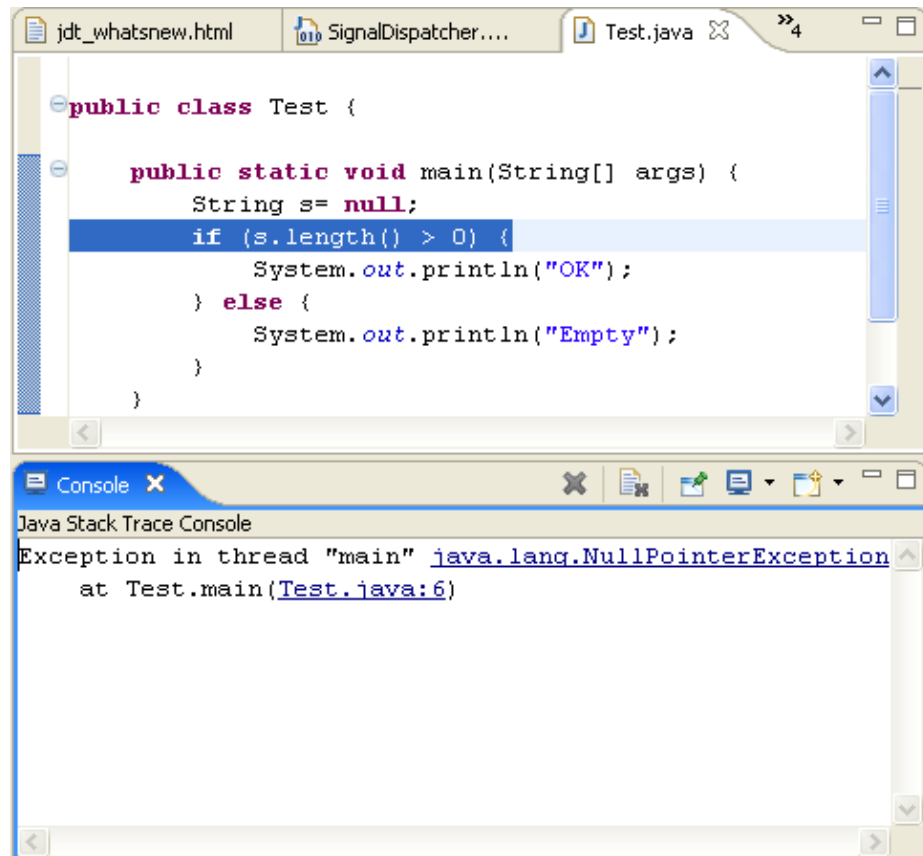
Watchpoints and method entry breakpoints	Double clicking in the Java editor ruler creates watchpoints on fields and method entry breakpoints on method declarations.
---	---

Locks and deadlocks	The locks owned by a thread as well as the lock a thread is waiting for can both be displayed inline in the Debug view by toggling the Show Monitors menu item in the Debug view drop-down menu. Threads and locks involved in a deadlock are highlighted in red.
----------------------------	--

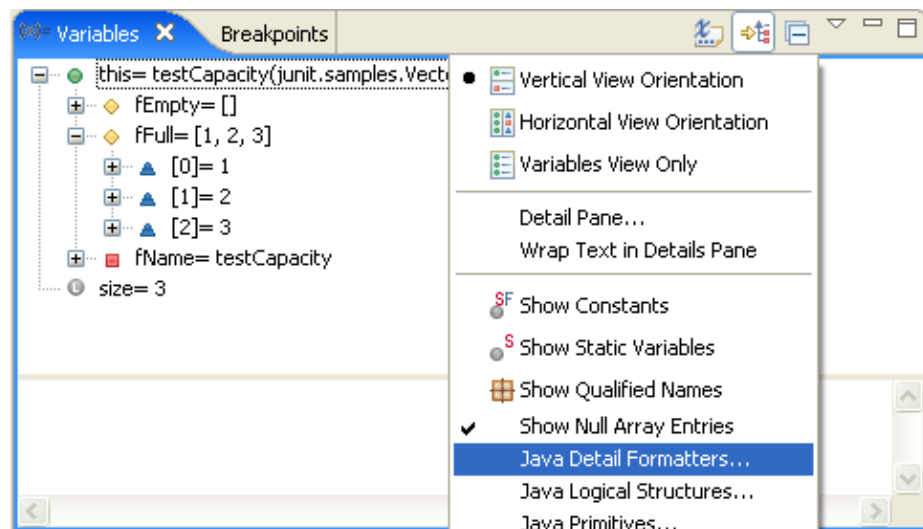


Navigating stack traces	Copy and paste a stack trace into the Java Stack Trace Console and use hyperlinks to navigate the trace. The Java Stack Trace Console can be opened from the Open Console drop-down menu in the Console view. Pasted stack traces can be formatted via the standard Format key binding.
--------------------------------	---

Basic tutorial



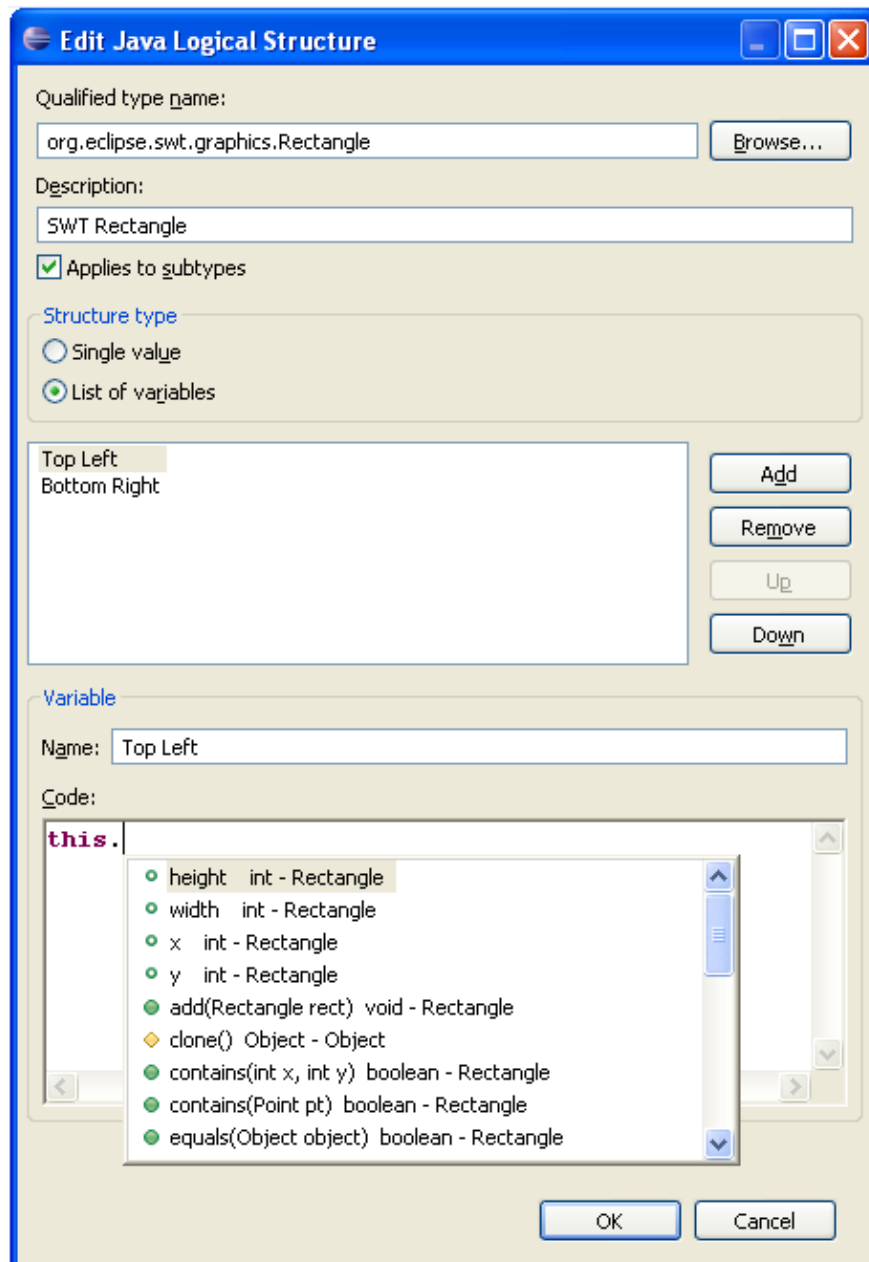
'toString()' inline The toString()–computed value of a variable can be displayed inline in the Variables view tree, as well as in the details area. The **Java Detail Formatters...** command in the view drop-down menu is used for configuring how this feature works.



**User-defined
logical
structures**

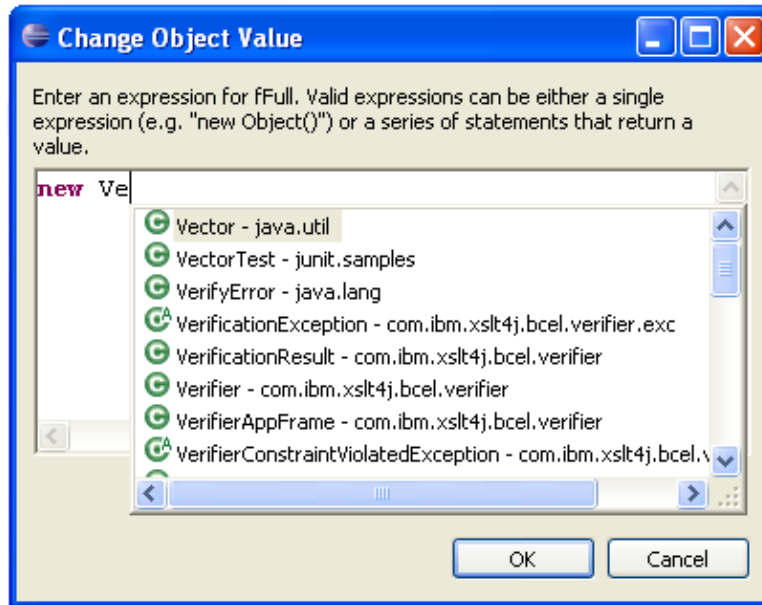
The Java debugger now lets you control what gets shown in the variables view for different types of objects. For example, collections can be displayed as a simple array of values, instead of the gory details on how that particular collection object is implemented.

This is done from the **Java > Debug > Logical Structures** preference page, where you associate with a specific class or interface either a single expression (for example, `this.toArray()`) or a series of named expressions. When the object is to be shown in the variables view, the expressions are evaluated to produce the values to display.



**Enhanced
variable
value
modification**

The Java debugger now lets you change the value of variables by entering an expression into either the **Change Value** dialog or into the details area of the variables view and pressing **Save**.

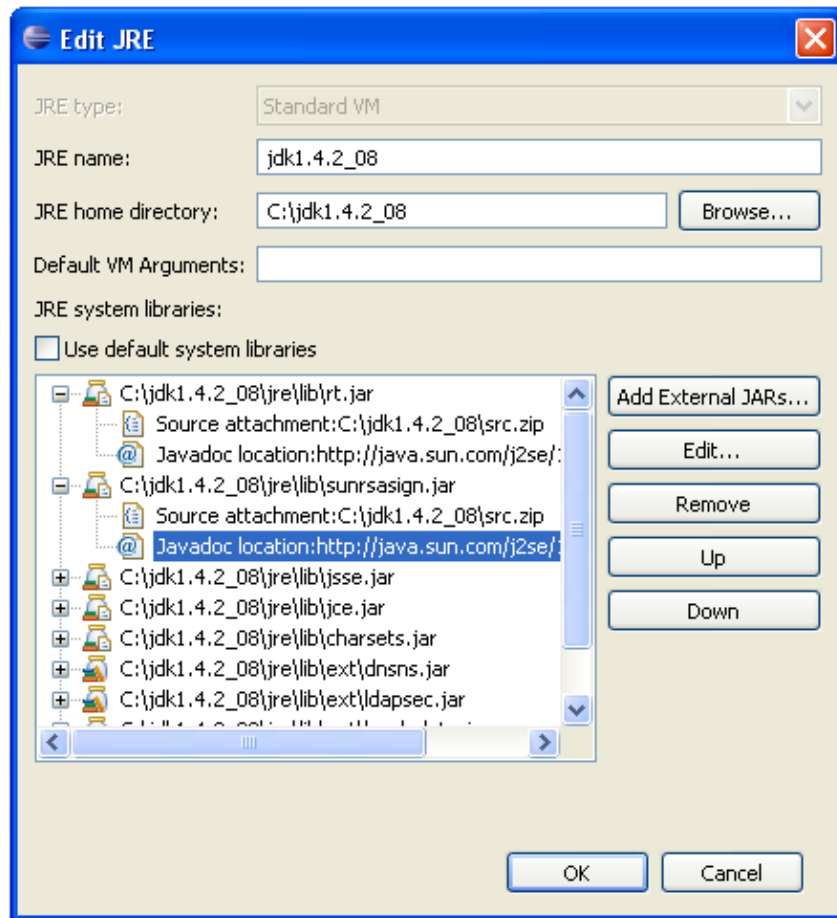


**Find
variable**

The Find Variable action in the Variables view allows you to type in the name of a variable you are looking for. As you type, the Variables view selects the next visible variable matching the entered text. As well, the Find variable dialog shows variables matching the text entered so far.

**Javadoc
attachments**

You can now associate a different Javadoc location with each JAR in a JRE's libraries.



Java Compiler

New Javadoc compiler settings

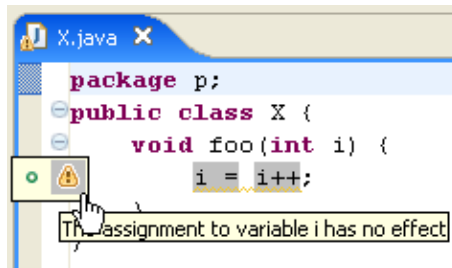
When Javadoc checking is enabled, you can configure it to

- warn when @see and @link tags reference deprecated elements
- warn when @see and @link tags reference elements that are not visible

The settings are on the **Java > Compiler > Javadoc** preference page.

**Assignment with
no effect diagnosis
for postfix
expression**

The optional diagnosis for *assignment with no effect* detects the case where a postfix expression is assigned to the same variable, e.g. `i = i++;`



**Serial Version
UID**

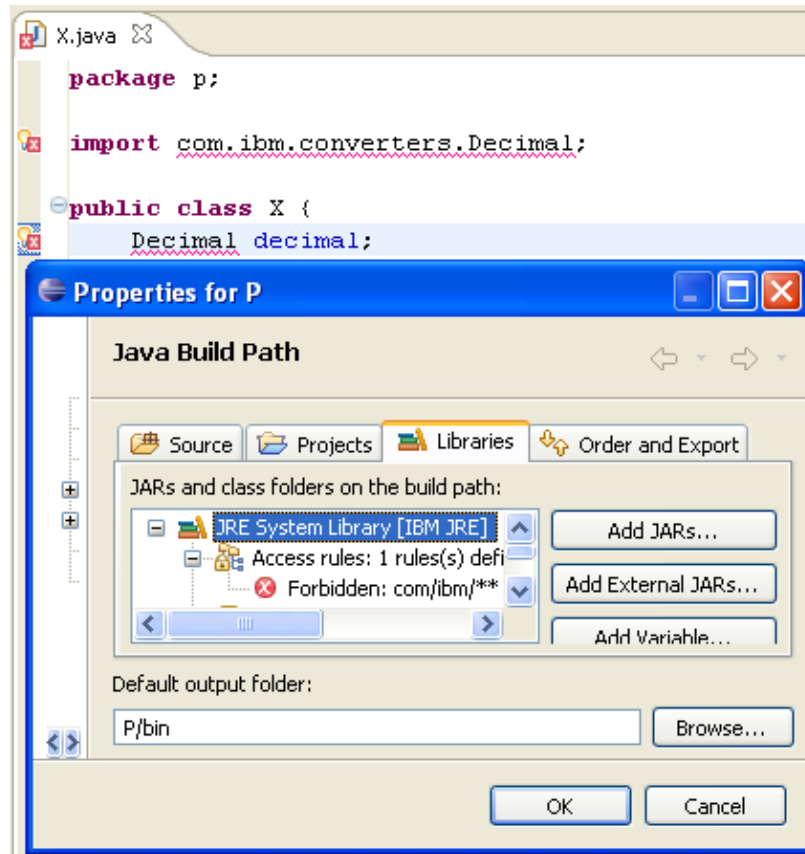
There is a new optional compiler diagnosis for serializable classes missing a declaration of a `serialVersionUID` field.

The preference setting can be found at **Java > Compiler > Errors/Warnings > Potential programming problems**

**Early detection of
references to
internal classes**

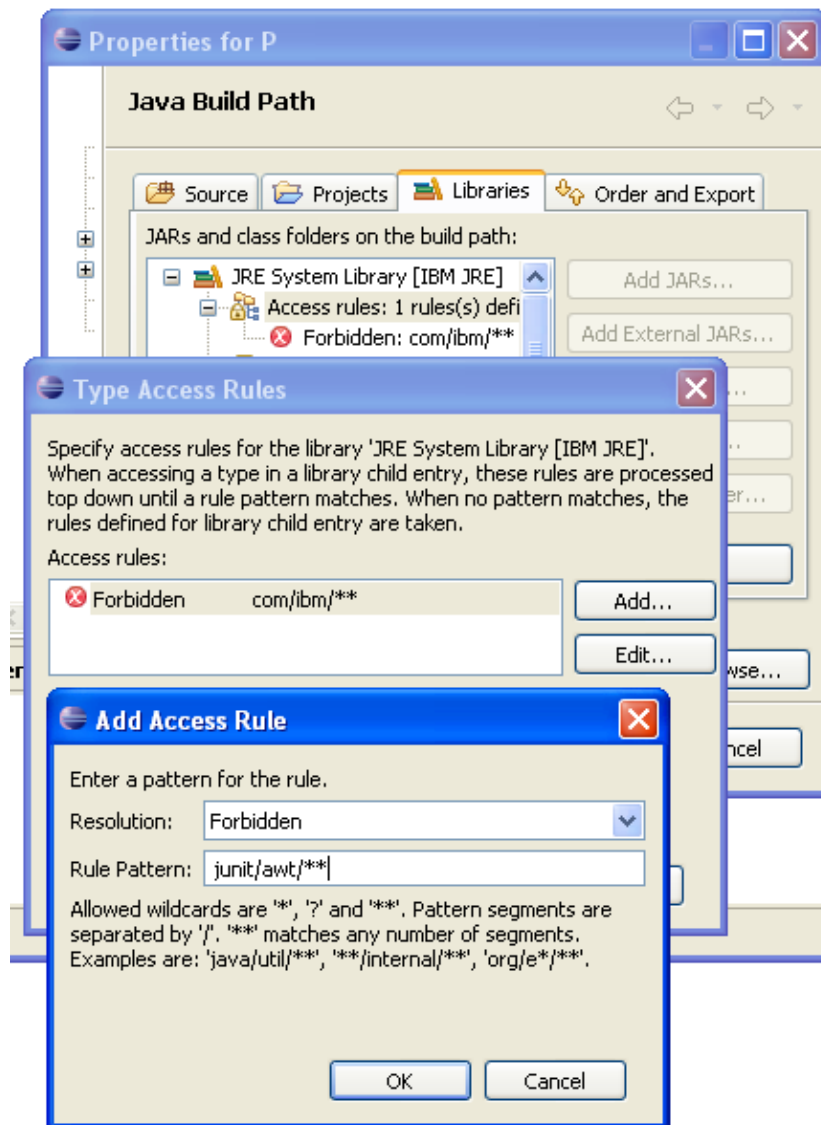
You can annotate library (and project) entries on the Java build path (**Properties > Java Build Path > Libraries**) to identify any internal packages that you want to avoid referencing directly from your code. For example, it's generally a bad idea to depend on any of the vendor-specific packages, like `com.ibm.*` or `com.sun.*`, commonly found in the J2SE libraries. Access restrictions are expressed with a combination of inclusion and exclusion rules on build path entries. The pattern syntax follows Ant fileset notation, and matches against the path to the class file. For example, using the pattern `com/ibm/**` as an exclusion rule would restrict access to all classes in the `com.ibm` package and its subpackages; using the pattern `org/eclipse/**/internal/**` as an exclusion rule would catch all classes to internal Eclipse packages. When you provide inclusion rules, everything matched by these rules is ok, and everything else is considered out of bounds.

The **Java > Compiler > Errors/Warnings > Deprecated and restricted API** preference setting lets you control whether errant references are flagged as errors or warnings (they are errors by default for forbidden reference and warnings for discouraged references).



Access rules on libraries and projects

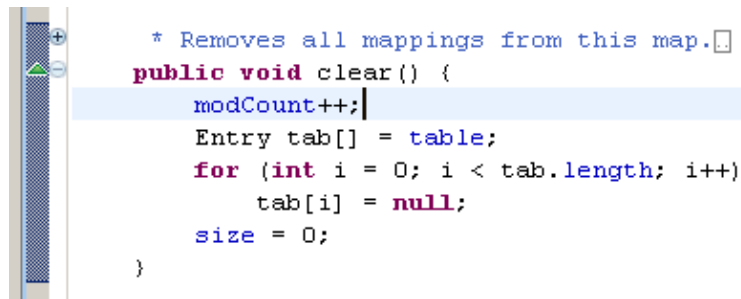
Access rules can be defined on referenced libraries and projects to explicitly allow/disallow/discourage access to specific types.



Java Editor

Improved folding icons and captions

When folding a Java element, the remaining line in the Java editor is the one containing the element's name. The first comment line is displayed for folded Javadoc comments. The new lightweight folding icons shown in the Java editor now differ from the override and implements indicators:

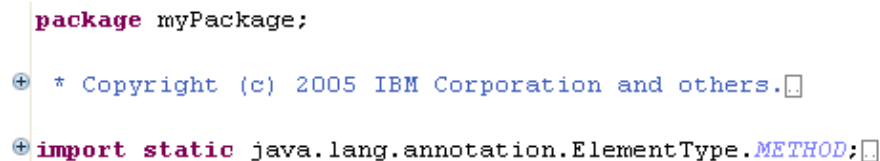


```

    * Removes all mappings from this map.
    public void clear() {
        modCount++;
        Entry tab[] = table;
        for (int i = 0; i < tab.length; i++)
            tab[i] = null;
        size = 0;
    }
  
```

Header comment folding

Header comments and copyright statements in Java source files can be folded:



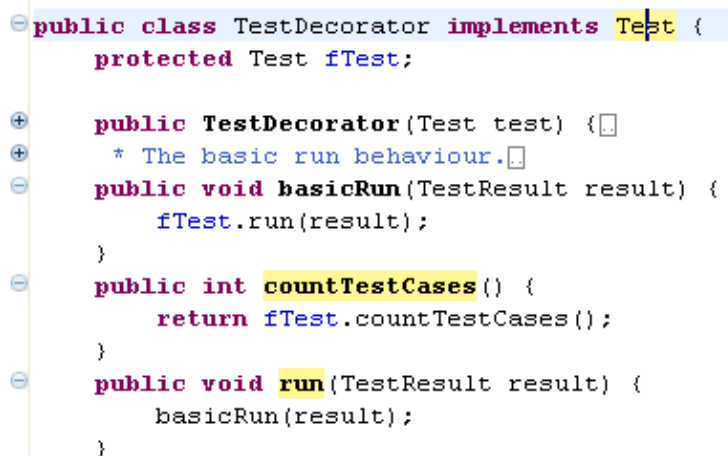
```

package myPackage;

+ * Copyright (c) 2005 IBM Corporation and others.
+ import static java.lang.annotation.ElementType.METHOD;
  
```

Mark occurrences of inherited methods

The Java editor can highlight all method declarations that implement or override methods inherited from the selected supertype. See the **Java > Editor > Mark Occurrences > Method implementing an interface** preference setting.



```

- public class TestDecorator implements Test {
    protected Test fTest;

    + public TestDecorator(Test test) {
    +     * The basic run behaviour.
    - public void basicRun(TestResult result) {
        fTest.run(result);
    }
    - public int countTestCases() {
        return fTest.countTestCases();
    }
    - public void run(TestResult result) {
        basicRun(result);
    }
  
```

New Occurrences Quick Menu

A context menu with occurrences searches can be opened in the Java editor by pressing Ctrl+Shift+U.

Note: Those who prefer the old behavior can reassign the above key sequence to the "Search All Occurrences in File" command.

Highlighting of deprecated class members in the Java editor

Deprecated class members are marked by advanced highlighting:

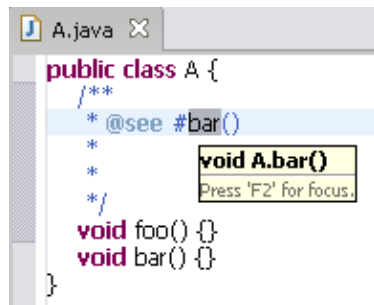
```
/**
 * @deprecated use {@link Test#setBar(long)} instead
 */
public void setFoo(long foo) {
    fFoo= foo;
}

public void update() {
    setFoo(System.currentTimeMillis());
}
```

This is configurable on the **Java > Editor > Syntax Coloring** preference page.

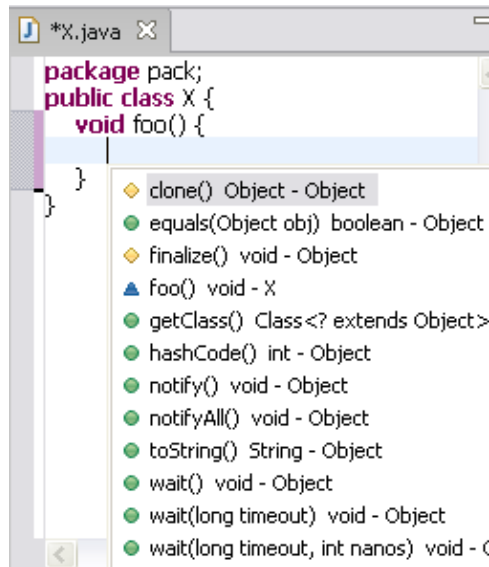
References in Javadoc

Eclipse now recognizes references to Java elements inside doc comments (i.e., @see, @link, @linkplain, @throws, @exception, @param or @value tags). This enables hover help and linking to the referenced Java element.



Improved completion on empty word

Java code completion on an empty word no longer automatically proposes all types visible at the completion location. You have to type the first character of the type to get a completion proposal.



**Tool tip
description
for Javadoc**

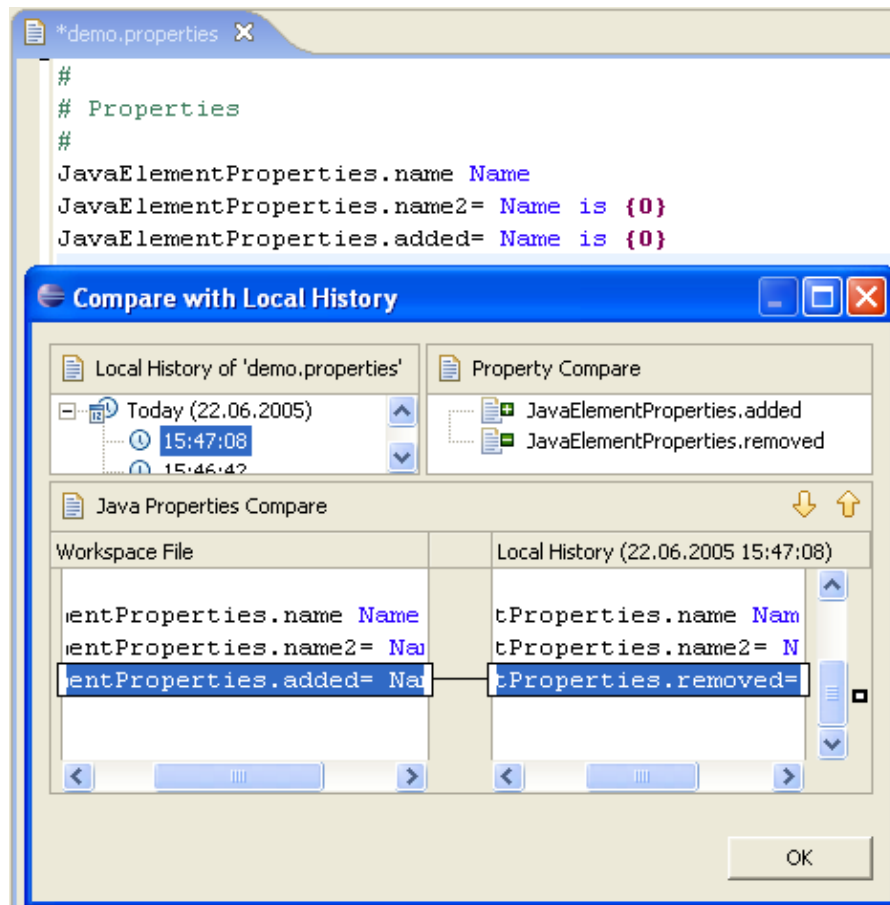
The Javadoc which is shown via **Edit > Show Tooltip Description (F2)** shows up in the SWT Browser widget.

**Move Lines
adjusts
indentation**

The **Move Lines (Alt+Up/Down)** and **Copy Lines (Ctrl+Alt+Up/Down)** commands automatically adjust the indentation of the selected lines as you move them inside the Java editor.

**Improved
Java
properties
file editor**

The editors for Java property files have been greatly improved. They offer syntax highlighting, improved double-clicking behavior, and a separate font preference. The syntax highlighting colors are adjusted from the **Java > Properties File Editor** preference page. Spell checking is also available, and Quick Fix (Ctrl+I) can be used to fix spelling problems.



Working with externalized strings

When you linger over a key for an externalized string in the Java editor, the associated externalized value is shown in a hover:

```
String s;
s= Messages.getString("test");
s= Messages.getString("test_undefined");
s= Messages.getString("test.long.key");
s= Messages.getString("SearchResultView.

Externalized Value:
Hello World
Press 'F2' for focus.
```

Ctrl+Click on it to navigate directly to the entry in the corresponding Java properties file:

```
String s;
s= Messages.getString("test");
s= Messages.getString("test_undefined");
s= Messages.getString("test.long.key");
s= Messages.getString("SearchResultView.
```

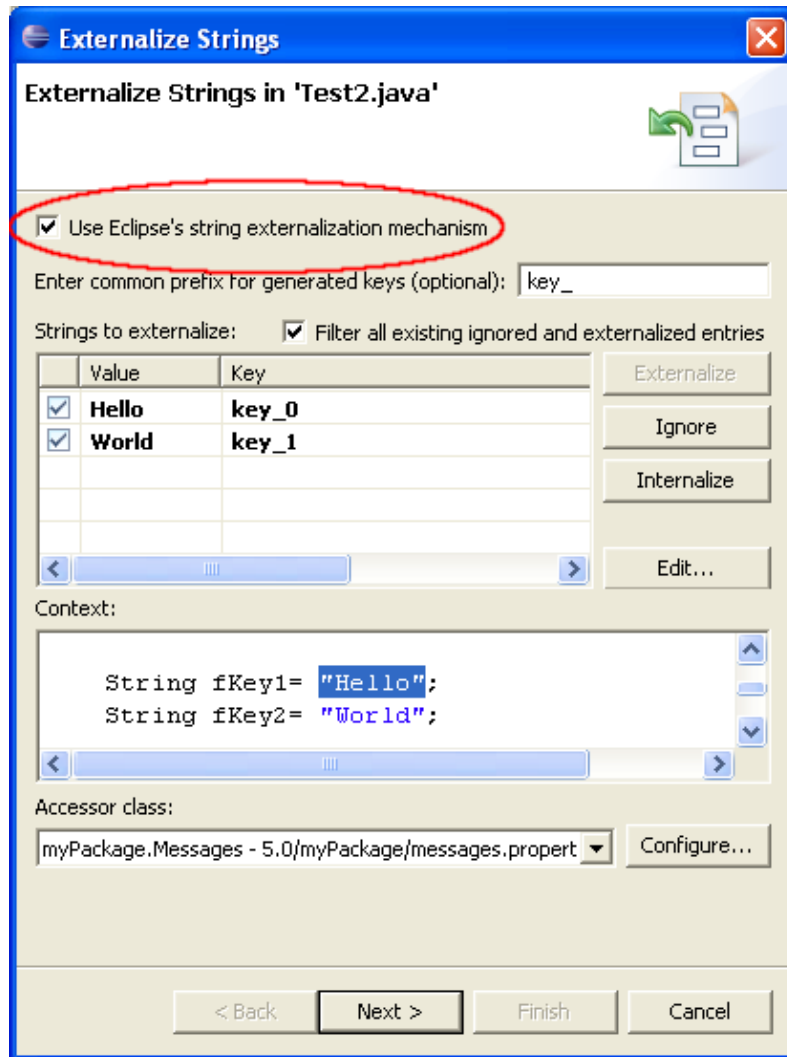
**Navigate
from
property key
in Properties
File editor to
its references**

Use **Navigate > Open (F3)** or **Ctrl+click** to navigate from a property key in the Properties File editor back to places in the code where the key is referenced.

```
TextViewer.error.bad_location.shift_2= TextViewer.shif
TextViewer.error.bad_location.verifyText= TextViewer.v
TextViewer.error.invalid_range= Invalid range argument
TextViewer.error.invalid_visible_region_1= Invalid vis
TextViewer.error.invalid_visible_region_2= Invalid vis
```

**Externalize
Strings
wizard
supports new
message
bundles**

The Externalize Strings wizard supports Eclipse's string externalization mechanism which is new with this release:



**New API to
create code
proposals
like in the
Java editor**

Implementing an editor for a Java-like language? Create your own code assist proposals similar to the ones proposed in the Java editor. Instantiate `CompletionProposalCollector` to get the same proposals as the Java editor, or subclass it to mix in your own proposals. Use `CompletionProposalLabelProvider` to get the images and labels right, and sort the proposals using `CompletionProposalComparator`.

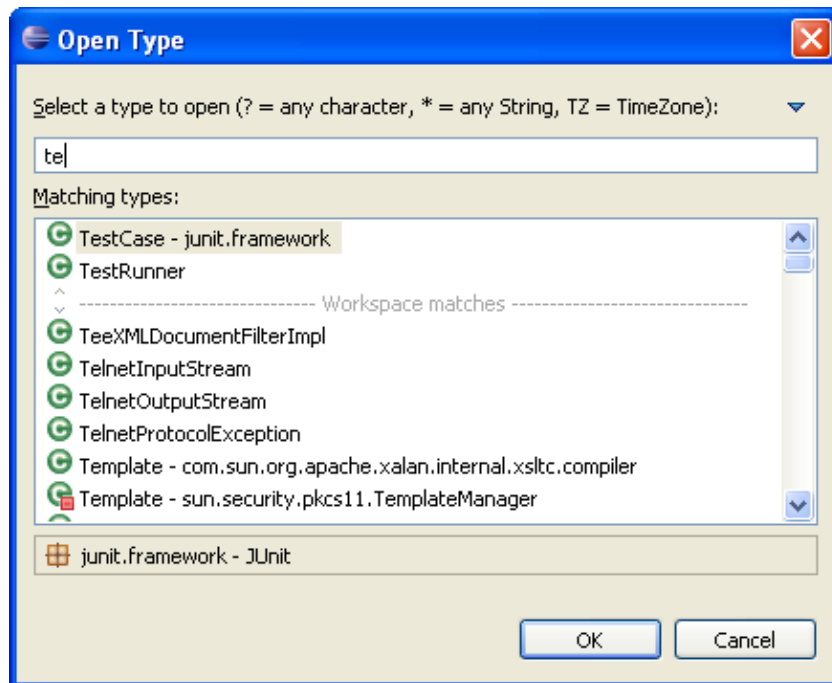
Package: `org.eclipse.jdt.ui.text.java` in the `org.eclipse.jdt.ui` plug-in.

General Java Tools

**New Open Type
dialog**

The Java Open Type dialog has been improved in a number of ways:

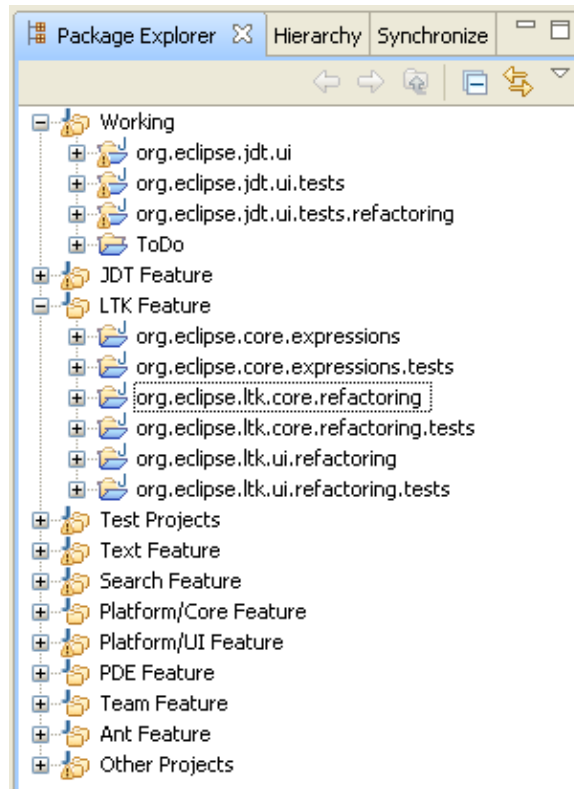
- There is now only a single list to select from.
- A history of recently opened types shows up first in the dialog; workspace types matching the pattern appear below the separator line.
- CamelCase pattern matching takes you to a type with fewer keystrokes. For example TZ matches `TimeZone` or `IOOBE` matches `IndexOutOfBoundsException`.
- The content of the dialog can further be constrained to a working set. The working set can be selected from the dialog's drop down menu.



There are major architectural changes under the hood as well. The types shown in the dialog are now found with a Java search engine query. This nets a saving of 4–6MB on a normal Eclipse development workspace over the memory–hungry approach used previously.

Organizing workspace with many projects

Use **Show > Working Sets** in the Package Explorer's view menu to enable a new mode that shows working sets as top level elements. This mode makes it much easier to manage workspaces containing lots of projects.



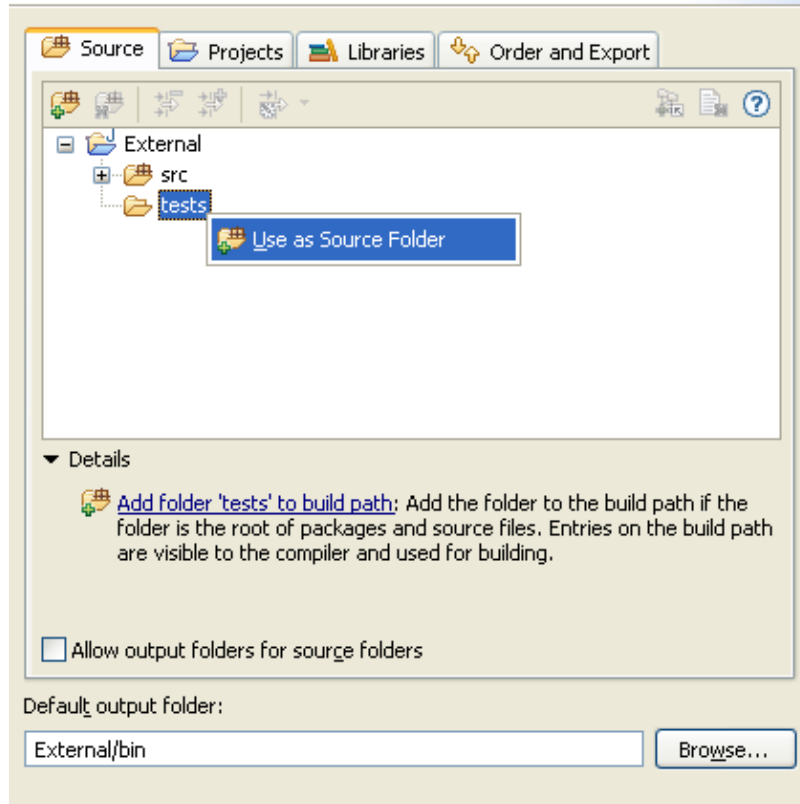
Use **Select Working Sets** from the Package Explorer's view menu to configure which working sets get shown. The dialog lets you create new Java working sets, define which working sets are shown and in what order. Working sets can also be rearranged directly in the Package Explorer using drag and drop and copy/paste.

Improved source folder page for new Java project wizard

An improved source folder configuration page in the Java project creation wizard assists you in creating projects from existing source. You can define source folder entries, include/exclude folders directly on the tree, and test the results of your action right away.

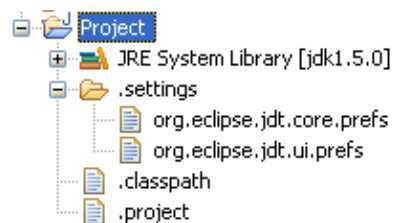
Java Settings

Define the Java build settings.

**Sharing Java project settings**

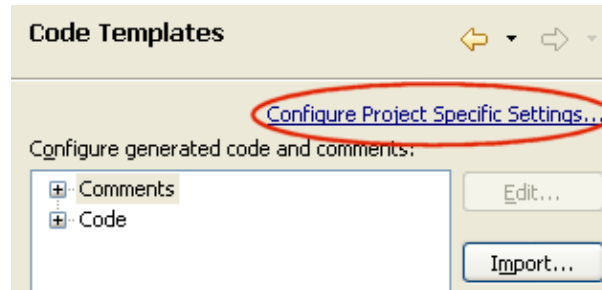
Each Java project can carry custom settings for compiler options and code style. These settings are stored in the project itself, and automatically applied when the project is loaded (or updated) from the repository.

Modifying the settings of a Java project via the UI automatically writes the settings to a file in the `.settings` directory. (The contents of the setting file are auto-generated, and not intended to be edited directly).



Navigate to project-specific settings

The preference pages for settings that are also configurable on a per-project basis offer a link to the project specific preference page.



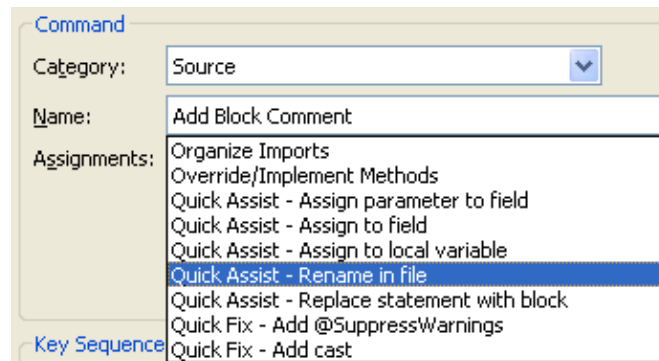
Javadoc locations stored in the .classpath file

The Javadoc locations that are attached to JAR files and class folders are stored in the .classpath file so they can be shared with the team. When 3.1 starts up, a background job will migrate all the previously internally stored locations to the .classpath file.

The Javadoc locations are used by 'Open External Javadoc' (CTRL + F2) and by the Javadoc wizard.

Shortcuts for quick assists and quick fixes

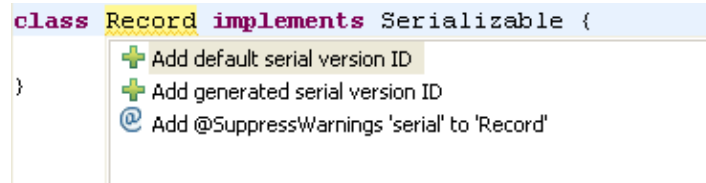
Some of the popular quick assists like *Rename In File* and *Assign To Local* can be invoked directly with **Ctrl+2 R** and **Ctrl+2 L**. Check the keys preference page for more quick fixes that support direct invocation.



New Quick Fixes

New Quick Fixes have been added for several Java compiler options, for example:

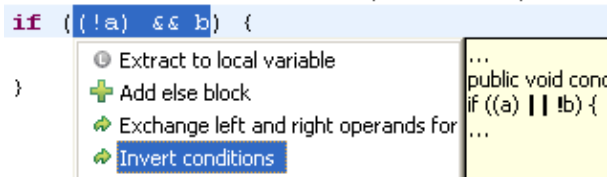
- Missing serial version ID:



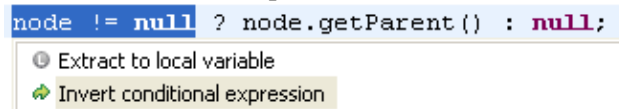
New Quick Assists

Several Quick Assists (Ctrl+I) have been added to the Java Editor:

- Invert boolean expressions:



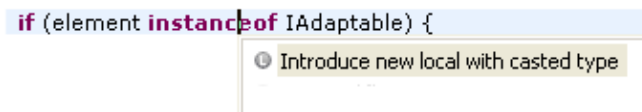
- Invert a conditional expression:



results in:

```
node == null ? null : node.getParent();
```

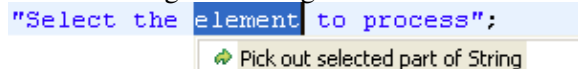
- Convert conditional expression (? operator) to if-else statement, or vice versa
- Introduce a new local variable after an instanceof check:



results in:

```
if (element instanceof IAdaptable) {
    IAdaptable adaptable = (IAdaptable) element;
}
```

- Break out single substring literal:

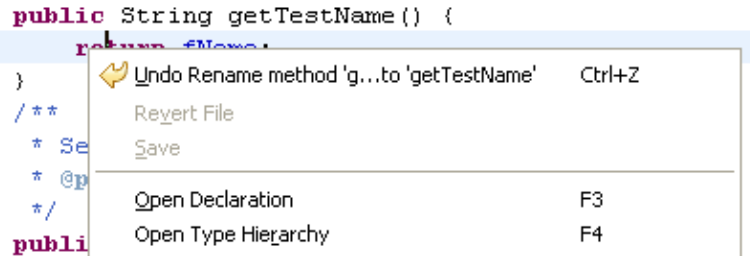


results in:

```
"Select the " + "element" + " to process";
```

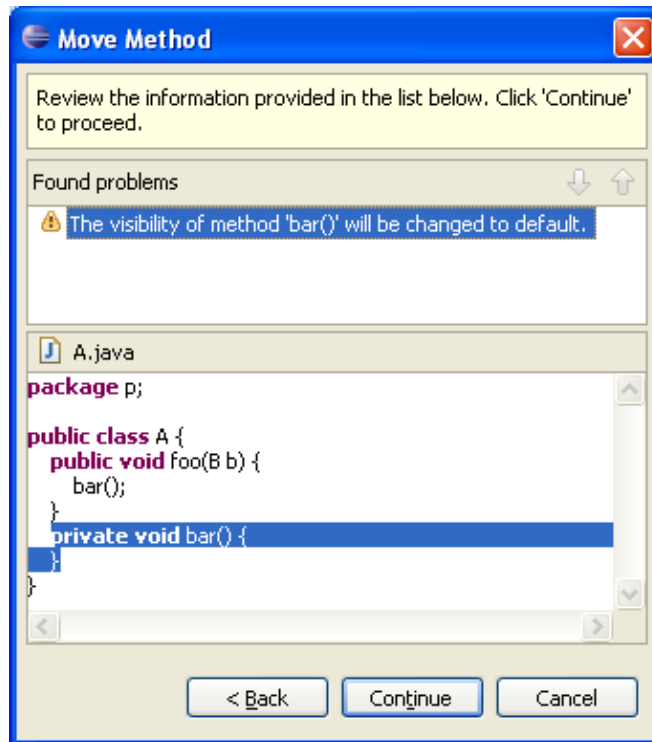
Refactoring Undo/Redo available from Edit menu

Refactoring Undo/Redo is now available from the Edit menu, and the separate Refactor Undo/Redo actions have been removed from the global menu bar. Additionally, refactoring Undo/Redo operations are now integrated with Java editor Undo/Redo, resulting in a more transparent undo story in the editor. For example, a refactoring triggered from within the editor is now undoable in the editor by simply pressing **Ctrl+Z**.



Member visibility adjustment

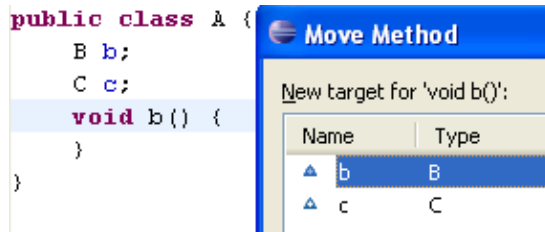
The refactoring commands Move method, Move Member Type to New File, Move Static member, Pull Up and Push Down automatically change the visibility of referenced fields, methods and types wherever necessary.



Move method refactoring

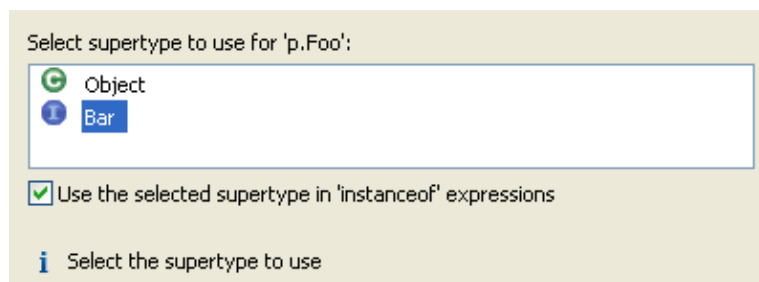
The **Refactor > Move** command has been extended to better support moving instance methods. New features include:

- The option to create a delegate method for compatibility
- Unreferenced fields can now also be method targets



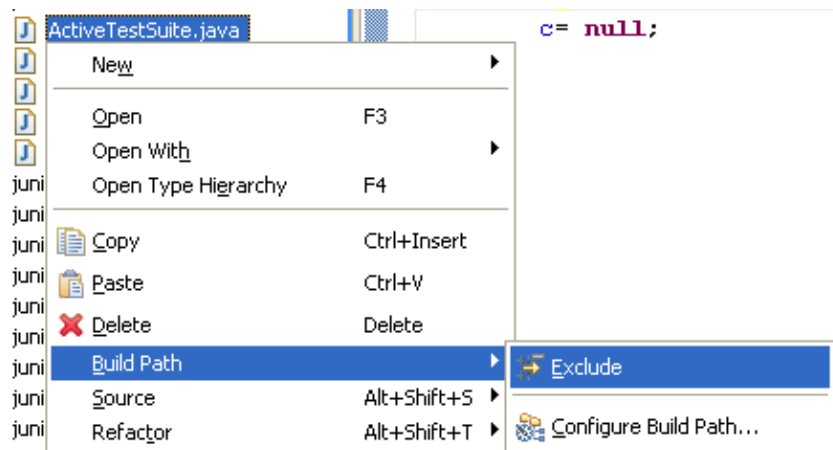
Use Supertype Where Possible refactoring

The Use Supertype Where Possible refactoring has been extended with a preference that specifies whether type occurrences in `instanceof` expressions should be updated:



Build path menu in the Package Explorer

The context menu of the Java Package Explorer has a new 'Build Path' menu entry, offering context-sensitive actions to modify the build path of a Java project. You can add/remove new source folders, archives and libraries, and include/exclude folders and files from a source folder:



New Eclipse default built-in

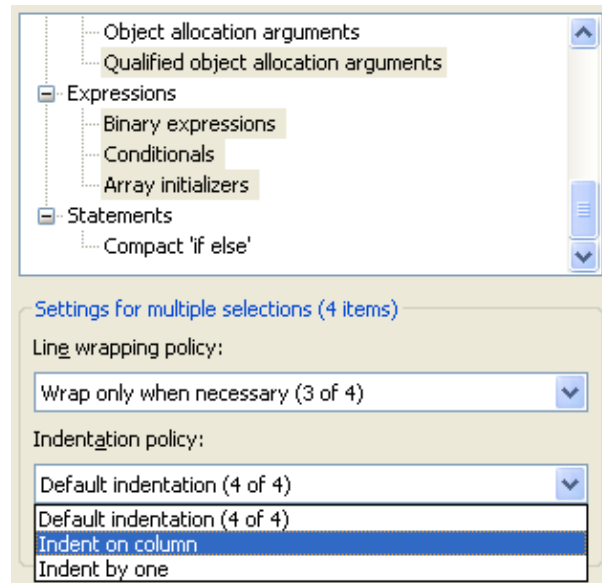
Although Eclipse's default 3.0 code formatter profile is named Java Conventions, formatting a file using this profile uses tabs for

formatter profile

indentation instead of spaces. A new profile named Eclipse has been added which reflects what the default formatter options have been all along, which uses tabs for indentation. To use true Java Convention settings, simply switch the formatter profile to Java Conventions using **Java > Code Style > Formatter** preference page.

Changing multiple line-wrap settings at once

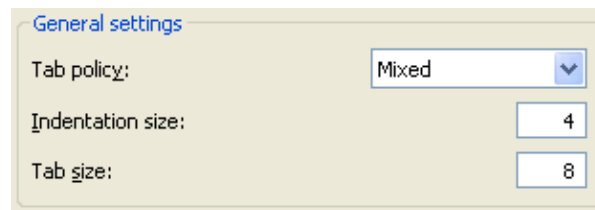
The Java code formatter page lets you change multiple line-wrap settings at once by multi-selecting in the tree and then changing settings:



Code formatter settings are on the **Java > Code Style > Formatter** preference page.

Mixed indentation settings

The Java formatter preferences allows the tab size to be configured independently from the indentation size (see the **Indentation** tab in your formatter profile):



For example, set **Tab size** to **8** and **Indentation size** to **4** to indent your source with four spaces. If you set the **Tab policy** to **Mixed**, every two indentation units will be replaced by a tab character.

Basic tutorial

Formatter profiles can be configured on the **Java > Code Style > Formatter** preference page.

Rerun failed tests first There's a new action in the JUnit view that allows you to rerun failing tests before any of the ones that were passing.

